CONTENTS

PREFACE

This book introduces applied numerical methods for engineering and science students in sophomore to senior levels; it targets the students of today who do not like or do not have time to derive and prove mathematical results. It can also serve as a reference to MATLAB applications for professional engineers and scientists, since many of the MATLAB codes presented after introducing each algorithm's basic ideas can easily be modified to solve similar problems even by those who do not know what is going on inside the MATLAB routines and the algorithms they use. Just as most drivers only have to know where to go and how to drive a car to get to their destinations, most users only have to know how to define the problems they want to solve using MATLAB and how to use the corresponding routines to solve their problems. We never deny that detailed knowledge about the algorithm (engine) of the program (car) is helpful for getting safely to the solution (destination); we only imply that one-time users of any MATLAB program or routine may use this book as well as the students who want to understand the underlying principle of each algorithm.

In this book, we focus on understanding the fundamental mathematical concepts and mastering problem-solving skills using numerical methods with the help of MATLAB and skip some tedious derivations. Obviously, basic concepts must be taught so that students can properly formulate the mathematics problems. Afterwards, students can directly use the MATLAB codes to solve practical problems. Almost every algorithm introduced in this book is followed by example MATLAB code with a friendly interface so that students can easily modify the code to solve real life problems. The selection of exercises follows the some philosophy of making the learning easy and practical. Students should be able to solve similar problems immediately after taking the class using the MATLAB codes we provide. For most students—and particularly nonmath majors—understanding how to use numerical tools correctly in solving their problems of interest is more important than studying lengthy proofs and derivations.

MATLAB is one of the most developed software packages available today. It provides many numerical methods and it is very easy to use, even for people without prior programming experience. We have supplemented MATLAB's built-in functions with more than 100 small MATLAB routines. Readers should find

these routines handy and useful. Some of these routines give better results for some problems than the built-in functions. Students are encouraged to develop their own routines following the examples.

The knowledge in this book is derived from the work of many eminent scientists, scholars, researchers, and MATLAB developers, all of whom we thank. We thank our colleagues, students, relatives, and friends for their support and encouragement. We thank the reviewers, whose comments were so helpful in tuning this book. We especially thank Senior Researcher Yong-Suk Park for his invaluable help in correction. We thank the editorial and production staff of John Wiley & Sons, Inc. including Editor Val Moliere and Production Editor Lisa VanHorn for their kind, efficient, and encouraging guide.

WON YOUNG YANG
WENWU CAO
TAE-SANG CHUNG
JOHN MORRIS

October 2004

1

MATLAB USAGE AND COMPUTATIONAL ERRORS

1.1 BASIC OPERATIONS OF MATLAB

MATLAB is a high-level software package with many built-in functions that make the learning of numerical methods much easier and more interesting. In this section we will introduce some basic operations that will enable you to learn the software and build your own programs for problem solving. In the workstation environment, you type "matlab" to start the program, while in the PC environment, you simply double-click the MATLAB icon.

Once you start the MATLAB program, a Command window will open with the MATLAB prompt >>. On the command line, you can type MATLAB commands, functions together with their input/output arguments, and the names of script files containing a block of statements to be executed at a time or functions defined by users. The MATLAB program files must have the extension name `***.m` to be executed in the MATLAB environment. If you want to create a new M-file or edit an existing file, you click File/New/M-file or File/Open in the top left corner of the main menu, find/select/load the file by double-clicking it, and then begin editing it in the Editor window. If the path of the file you want to run is not listed in the MATLAB search path, the file name will not be recognized by MATLAB. In such cases, you need to add the path to the MATLAB-path list by clicking the menu 'File/Set_Path' in the Command window, clicking the 'Add_Folder' button, browsing/clicking the folder name, and finally clicking the SAVE button and the Close button. The *lookfor* command is available to help you find the MATLAB commands/functions which are related with a job you

---

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

want to be done. The *help* command helps you know the usage of a particular command/function. You may type directly in the Command window

```
>>lookfor repeat    or    >>help for
```

to find the MATLAB commands in connection with 'repeat' or to obtain information about the "for loop".

1.1.1 Input/Output of Data from MATLAB Command Window

MATLAB remembers all input data in a session (anything entered through direct keyboard input or running a script file) until the command 'clear()' is given or you exit MATLAB.

One of the many features of MATLAB is that it enables us to deal with the vectors/matrices in the same way as scalars. For instance, to input the matrices/vectors,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 3 \\ -2 \\ 1 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & -2 & 3 & -4 \end{bmatrix}$$

type in the MATLAB Command window as below:

```
>>A = [1 2 3;4 5 6]
  A = 1    2    3
      4    5    6
>>B = [3;-2;1]; %put the semicolon at the end of the statement to suppress
                 the result printout onto the screen
>>C = [1 -2  3 -4]
```

At the end of the statement, press <Enter> if you want to check the result of executing the statement immediately. Otherwise, type a semicolon ";" before pressing <Enter> so that your window will not be overloaded by a long display of results.

1.1.2 Input/Output of Data Through Files

MATLAB can handle two types of data files. One is the binary format mat-files named ***.mat. This kind of file can preserve the values of more than one variable, but will be handled only in the MATLAB environment and cannot be shared with other programming environments. The other is the ASCII dat-files named ***.dat, which can be shared with other programming environments, but preserve the values of only one variable.

Below are a few sample statements for storing some data into a mat-file in the current directory and reading the data back from the mat-file:

```
>>save ABC A B C %store the values of A,B,C into the file 'ABC.mat'
>>clear A C %clear the memory of MATLAB about A,C
```

```
>>A %what is the value of A?
   ??? Undefined function or variable 'A'
>>load ABC A C %read the values of A,C from the file 'ABC.mat'
>>A %the value of A
   A = 1    2    3
       4    5    6
```

If you want to store the data into an ASCII dat-file (in the current directory), make the filename the same as the name of the data and type '`/ascii`' at the end of the *save* statement.

```
>>save B.dat B /ascii
```

However, with the save/load commands into/from a dat-file, the value of only one variable having the lowercase name can be saved/loaded, a scalar or a vector/matrix. Besides, non-numeric data cannot be handled by using a dat-file. If you save a string data into a dat-file, its ASCII code will be saved. If a dat-file is constructed to have a data matrix in other environments than MATLAB, every line (row) of the file must have the same number of columns. If you want to read the data from the dat-file in MATLAB, just type the (lowercase) filename `***`.dat after '`load`', which will also be recognized as the name of the data contained in the dat-file.

```
>>load b.dat %read the value of variable b from the ascii file 'b.dat'
```

On the MATLAB command line, you can type '`nm112`' to run the following M-file '`nm112.m`' consisting of several file input(save)/output(load) statements. Then you will see the effects of the individual statements from the running results appearing on the screen.

```
%nm112.m
clear
A = [1 2 3;4 5 6]
B = [3;-2;1];
C(2) = 2; C(4) = 4
disp('Press any key to see the input/output through Files')
save ABC A B C %save A,B & C as a MAT-file named 'ABC.mat'
clear('A','C') %remove the memory about A and C
load ABC A C %read MAT-file to recollect the memory about A and C
save B.dat B /ascii %save B as an ASCII-file named 'b.dat'
clear B
load b.dat %read ASCII-file to recollect the memory about b
b
x = input('Enter x:')
format short e
x
format rat, x
format long, x
format short, x
```

1.1.3 Input/Output of Data Using Keyboard

The command '*input*' enables the user to input some data via the keyboard. For example,

```
>>x = input('Enter x: ')
    Enter x: 1/3
    x = 0.3333
```

Note that the fraction 1/3 is a nonterminating decimal number, but only four digits after the decimal point are displayed as the result of executing the above command. This is a choice of formatting in MATLAB. One may choose to display more decimal places by using the command 'format', which can make a fraction show up as a fraction, as a decimal number with more digits, or even in an exponential form of a normalized number times 10 to the power of some integer. For instance:

```
>>format rat %as a rational number
>>x
  x = 1/3
>>format long %as a decimal number with 14 digits
>>x
  x = 0.33333333333333
>>format long e %as a long exponential form
>>x
  x = 3.333333333333333e-001
>>format hex %as a hexadecimal form as represented/stored in memory
>>x
  x = 3fd5555555555555
>>format short e %as a short exponential form
>>x
  x = 3.3333e-001
>>format short %back to a short form (default)
>>x
  x = 0.3333
```

Note that the number of displayed digits is not the actual number of significant digits of the value stored in computer memory. This point will be made clear in Section 1.2.1.

There are other ways of displaying the value of a variable and a string on the screen than typing the name of the variable. Two useful commands are 'disp()' and 'fprintf()'. The former displays the value of a variable or a string without 'x = ' or 'ans = '; the latter displays the values of several variables in a specified format and with explanatory/cosmetic strings. For example:

```
>>disp('The value of x = '),disp(x)
     %disp('string_to_display' or variable_name)
  The value of x = 0.3333
```

Table 1.1 summarizes the type specifiers and special characters that are used in 'fprintf()' statements.

Below is a program that uses the command '*input*' so that the user could input some data via the keyboard. If we run the program, it gets a value of the

Table 1.1 Type Specifiers and Special Characters Used in `fprintf()` Statements

Type Specifier	Printing Form: fprintf('**format string**', variables_to_be_printed,..)	Special Character	Meaning
`%c`	Character type	`\n`	New line
`%s`	String type	`\t`	Tab
`%d`	Decimal integer number type	`\b`	Backspace
`%f`	Floating point number type	`\r`	CR return
`%e`	Decimal exponential type	`\f`	Form feed
`%x`	Hexadecimal integer number	`%%`	`%`
`%bx`	Floating number in 16 hexadecimal digits(64 bits)	`''`	`'`

temperature in Fahrenheit [°F] via the keyboard from the user, converts it into the temperature in Centigrade [°C] and then prints the results with some remarks both onto the screen and into a data file named 'nm113.dat'.

```
%nm113.m
f = input('Input the temperature in Fahrenheit[F]:');
c = 5/9*(f-32);
fprintf('%5.2f(in Fahrenheit) is %5.2f(in Centigrade).\n',f,c)
fid=fopen('nm113.dat', 'w');
fprintf(fid, '%5.2f(Fahrenheit) is %5.2f(Centigrade).\n',f,c);
fclose(fid);
```

In case you want the keyboard input to be recognized as a string, you should add the character `'s'` as the second input argument.

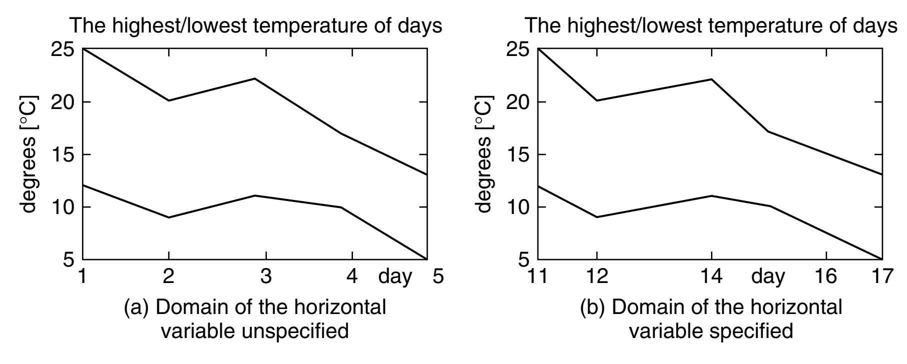
```
>>ans = input('Answer <yes> or <no>: ','s')
```

1.1.4 2-D Graphic Input/Output

How do we plot the value(s) of a vector or an array? Suppose that data reflecting the highest/lowest temperatures for 5 days are stored as a 5 × 2 array in an ASCII file named '`temp.dat`'.

The job of the MATLAB program "`nm114_1.m`" is to plot these data. Running the program yields the graph shown in Fig. 1.1a. Note that the first line is a comment about the name and the functional objective of the program(file), and the fourth and fifth lines are auxiliary statements that designate the graph title and units of the vertical/horizontal axis; only the second & third lines are indispensable in drawing the colored graph. We need only a few MATLAB statements for this artwork, which shows the power of MATLAB.

```
%nm114_1: plot the data of a 5x2 array stored in "temp.dat"
load temp.dat
clf, plot(temp) %clear any existent figure and plot
title('the highest/lowest temperature of these days')
ylabel('degrees[C]'), xlabel('day')
```



(a) Domain of the horizontal variable unspecified

(b) Domain of the horizontal variable specified

Figure 1.1 Plot of a 5 × 2 matrix data representing the highest/lowest temperature.

Here are several things to keep in mind.

- The command `plot()` reads along the columns of the 5 × 2 array data given as its input argument and recognizes each column as the value of a vector.
- MATLAB assumes the domain of the horizontal variable to be [1 2 .. 5] by default, where 5 equals the length of the vector to be plotted (see Fig. 1.1a).
- The graph is constructed by connecting the data points with the straight lines and is piecewise-linear, while it looks like a curve as the data points are densely collected. Note that the graph can be plotted as points in various forms according to the optional input argument described in Table 1.2.

(Q1) Suppose the data in the array named 'temp' are the highest/lowest temperatures measured on the 11th,12th,14th,16th, and 17th days, respectively. How should we modify the above program to have the actual days shown on the horizontal axis?

(A1) Just make the day vector [11 12 14 16 17] and use it as the first input argument of the `plot()` command.

```
>>days = [11 12 14 16 17]
>>plot(days,temp)
```

Executing these statements, we obtain the graph in Fig. 1.1b.

(Q2) What statements should be added to change the ranges of the horizontal/vertical axes into 10–20 and 0–30, respectively, and draw the grid on the graph?

Table 1.2 Graphic Line Specifications Used in the `plot()` Command

Line Type	Point Type (Marker Symbol)			Color	
`-` solid line	`.` (dot)	`+` (plus)	`*` (asterisk)	`r` : red	`m` : magenta
`:` dotted line	`^` : △	`>` : >	`o` (circle)	`g` : green	`y` : yellow
`--` dashed line	`p` : ☆	`v` : ▽	`x` : x-mark	`b` : blue	`c` : cyan (sky blue)
`-.` dash-dot	`d` : ◇	`<` : <	`s` : □	`k` : black	

(A2) `>>axis([10 20 0 30]), grid on`
`>>plot(days,temp)`

(Q3) How do we make the scales of the horizontal/vertical axes equal so that a circle appears round, not like an ellipse?

(A3) `>>axis('equal')`

(Q4) How do we have another graph overlapped onto an existing graph?

(A4) If you use the ‘`hold on`’ command after plotting the first graph, any following graphs in the same section will be overlapped onto the existing one(s) rather than plotted newly. For example:

```
>>hold on, plot(days,temp(:,1),'b*', days,temp(:,2),'ro')
```

This will be good until you issue the command ‘`hold off`’ or clear all the graphs in the graphic window by using the ‘`clf`’ command.

Sometimes we need to see the interrelationship between two variables. Suppose we want to plot the lowest/highest temperature, respectively, along the horizontal/vertical axis in order to grasp the relationship between them. Let us try using the following command:

```
>>plot(temp(:,1),temp(:,2),'kx') % temp(:,2) vs. temp(:,1) in black 'x'
```

This will produce a pointwise graph, which is fine. But, if you replace the third input argument by ‘`b:`’ or just omit it to draw a piecewise-linear graph connecting the data points as Fig. 1.2a, the graphic result looks clumsy, because the data on the horizontal axis are not arranged in ascending or descending order. The graph will look better if you sort the data on the horizontal axis and also the data on the vertical axis accordingly and then plot the relationship in the piecewise-linear style by typing the MATLAB commands as follows:

```
>>[temp1,I] = sort(temp(:,1)); temp2 = temp(I,2);
>>plot(temp1,temp2)
```

The graph obtained by using these commands is shown in Fig.1.2b, which looks more informative than Fig.1.2a.

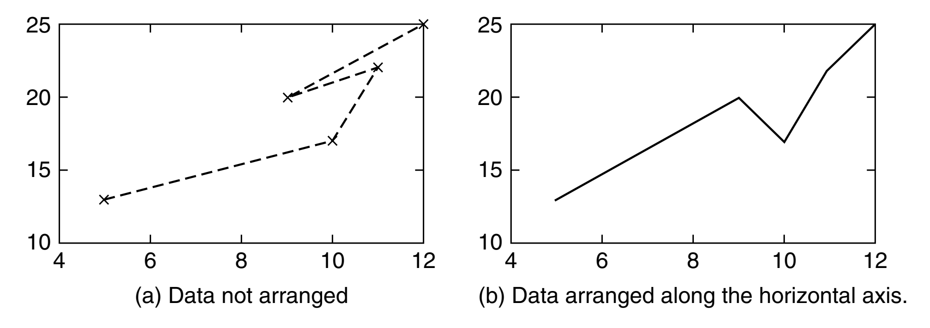


Figure 1.2 Examples of graphs obtained using the `plot()` command.

We can also use the `plot()` command to draw a circle.

```
>>r = 1; th = [0:0.01:2]*pi; % [0:0.01:2] makes [0 0.01 0.02 .. 2]
>>plot(r*cos(th),r*sin(th))
>>plot(r*exp(j*th)) %alternatively,
```

Note that the `plot()` command with a sequence of complex numbers as its first input argument plots the real/imaginary parts along the horizontal/vertical axis.

The `polar()` command plots the phase (in radians)/magnitude given as its first/second input argument, respectively (see Fig.1.3a).

```
>>polar(th,exp(-th)) %polar plot of a spiral
```

Several other plotting commands, such as `semilogx()`, `semilogy()`, `loglog()`, `stairs()`, `stem()`, `bar()`/`barh()`, and `hist()`, may be used to draw various graphs (shown in Figs.1.3 and 1.4). Readers may use the '`help`' command to get the detailed usage of each one and try running the following MATLAB program 'nm114_2.m'.

```
%nm114_2: plot several types of graph
th = [0: .02:1]*pi;
subplot(221), polar(th,exp(-th))
subplot(222), semilogx(exp(th))
subplot(223), semilogy(exp(th))
subplot(224), loglog(exp(th))
pause, clf
subplot(221), stairs([1 3 2 0])
subplot(222), stem([1 3 2 0])
subplot(223), bar([2 3; 4 5])
subplot(224), barh([2 3; 4 5])
pause, clf
y = [0.3 0.9 1.6 2.7 3 2.4];
subplot(221), hist(y,3)
subplot(222), hist(y,0.5 + [0 1 2])
```

Moreover, the commands `sprintf()`, `text()`, and `gtext()` are used for combining supplementary statements with the value(s) of one or more variables to construct a string and printing it at a certain location on the existing graph. For instance, let us try the following statements in the MATLAB Command window:

```
>>f = 1./[1:10]; plot(f)
>>n = 3; [s,errmsg] = sprintf('f(%1d) = %5.2f',n,f(n))
>>text(3,f(3),s) %writes the text string at the point (3,f(3))
>>gtext('f(x) = 1/x') %writes the input string at point clicked by mouse
```

The command `ginput()` allows you to obtain the coordinates of a point by clicking the mouse button on the existent graph. Let us try the following

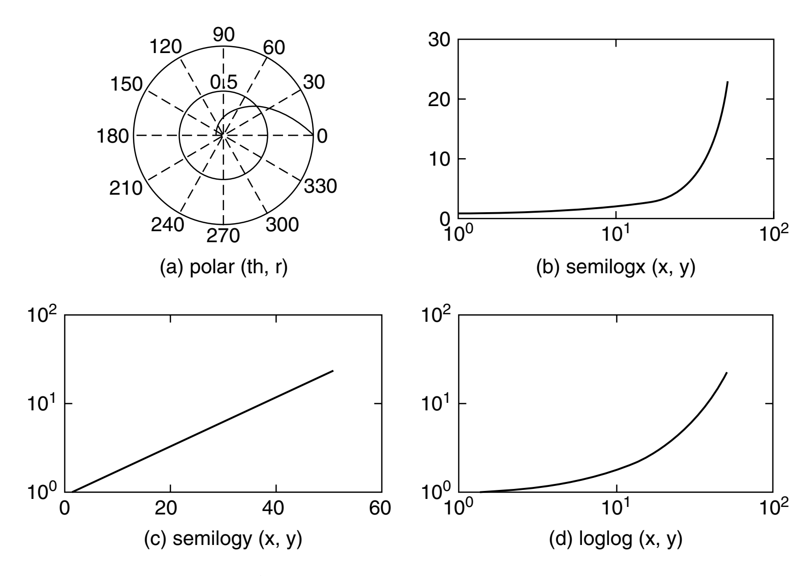


Figure 1.3 Graphs drawn by various graphic commands.

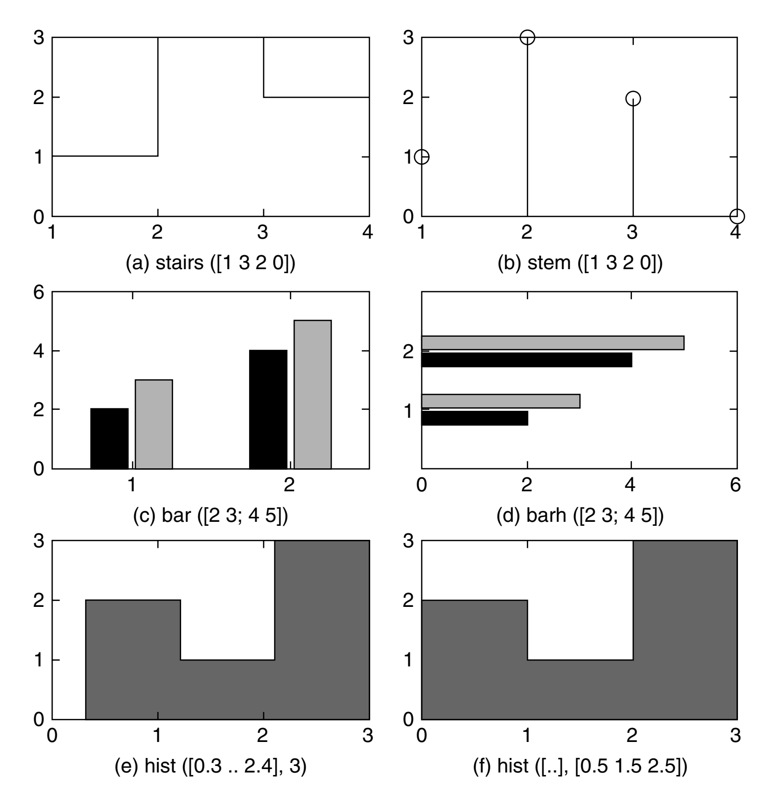


Figure 1.4 Graphs drawn by various graphic commands.

commands:

```
>>[x,y,butkey] = ginput %get the x,y coordinates & # of the mouse button
      or ascii code of the key pressed till pressing the ENTER key
>>[x,y,butkey] = ginput(n) %repeat the same job for up to n points clicked
```

1.1.5 3-D Graphic Output

MATLAB has several 3-D graphic plotting commands such as `plot3()`, `mesh()`, and `contour()`. `plot3()` plots a 2-D valued-function of a scalar-valued variable; `mesh()/contour()` plots a scalar valued-function of a 2-D variable in a mesh/contour-like style, respectively.

Readers are recommended to use the `help` command for detailed usage of each command. Try running the MATLAB program 'nm115.m' to see what figures will appear (Figs.1.5 and 1.6).

```
%nm115: to plot 3D graphs
t = 0:pi/50:6*pi;
expt = exp(-0.1*t);
xt = expt.*cos(t); yt = expt.*sin(t);
%dividing the screen into 2 x 2 sections
subplot(221), plot3(xt, yt, t), grid on %helix
subplot(222), plot3(xt, yt, t), grid on, view([0  0 1])
subplot(223), plot3(t, xt, yt), grid on, view([1 -3 1])
subplot(224), plot3(t, yt, xt), grid on, view([0 -3 0])
pause, clf
x = -2:.1:2;  y = -2:.1:2;
[X,Y] = meshgrid(x,y); Z = X.^2 + Y.^2;
subplot(221), mesh(X,Y,Z), grid on %[azimuth,elevation] = [-37.5,30]
subplot(222), mesh(X,Y,Z), view([0,20]), grid on
pause, view([30,30])
subplot(223), contour(X,Y,Z)
subplot(224), contour(X,Y,Z,[.5,2,4.5])
```

1.1.6 Mathematical Functions

Mathematical functions and special reserved constants/variables defined in MATLAB are listed in Table 1.3.

MATLAB also allows us to define our own function and store it in a file named after the function name so that it can be used as if it were a built-in function. For instance, we can define a scalar-valued function:

$$f_1(x) = 1/(1 + 8x^2)$$

and a vector-valued function

$$f_{49}(x) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_1^2 + 4x_2^2 - 5 \\ 2x_1^2 - 2x_1 - 3x_2 - 2.5 \end{bmatrix}$$

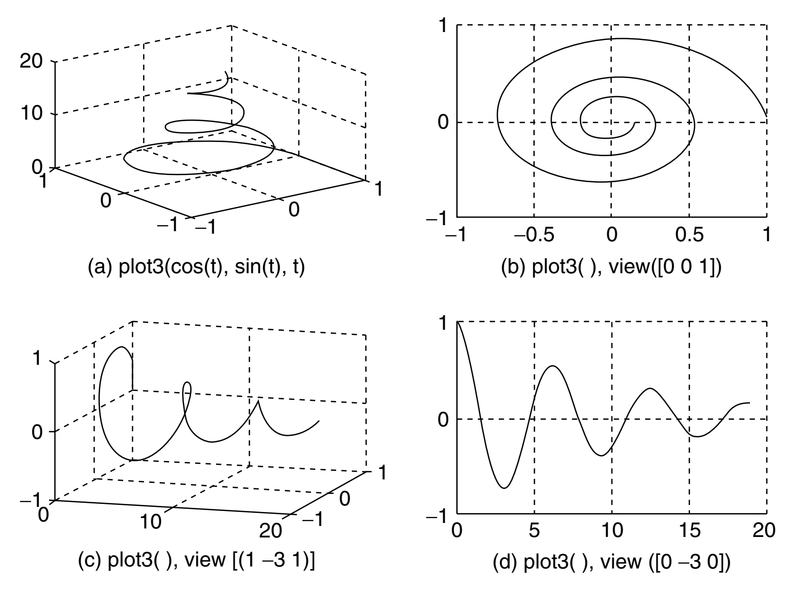


Figure 1.5 Graphs drawn by the `plot3()` command with different views.

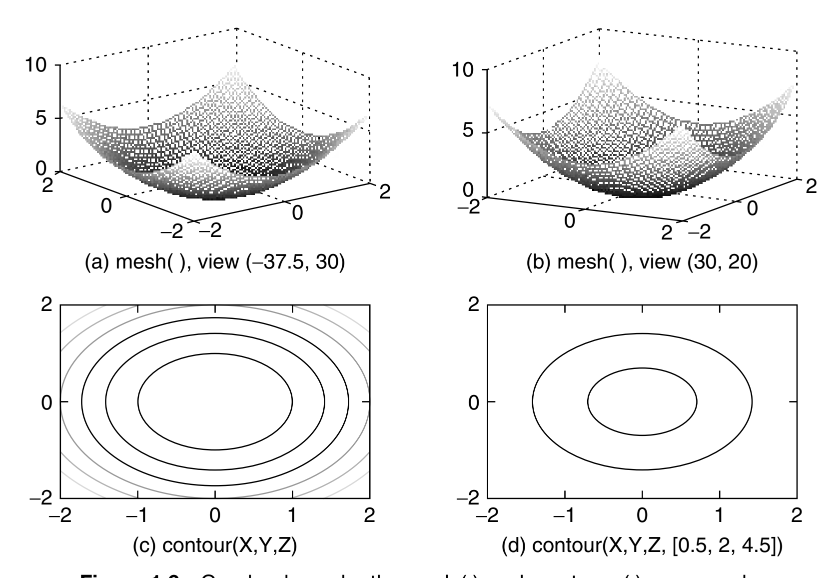


Figure 1.6 Graphs drawn by the `mesh()` and `contour()` commands.

as follows.

`function y = f1(x)`
`y = 1./(1+8*x.^2);`
	`function y = f49(x)`
	`y(1) = x(1)*x(1)+4*x(2)*x(2) -5;`
	`y(2) = 2*x(1)*x(1)-2*x(1)-3*x(2) -2.5;`

Table 1.3 Functions and Variables Inside MATLAB

Function	Remark	Function	Remark
`cos(x)`		`exp(x)`	Exponential function
`sin(x)`		`log(x)`	Natural logarithm
`tan(x)`		`log10(x)`	Common logarithm
`acos(x)`	$\cos^{-1}(x)$	`abs(x)`	Absolute value
`asin(x)`	$\sin^{-1}(x)$	`angle(x)`	Phase of a complex number [rad]
`atan(x)`	$-\pi/2 \le \tan^{-1}(x) \le \pi/2$	`sqrt(x)`	Square root
`atan2(y,x)`	$-\pi \le \tan^{-1}(y,x) \le \pi$	`real(x)`	Real part
`cosh(x)`	$(e^x + e^{-x})/2$	`imag(x)`	Imaginary part
`sinh(x)`	$(e^x - e^{-x})/2$	`conj(x)`	Complex conjugate
`tanh(x)`	$(e^x - e^{-x})/(e^x + e^{-x})$	`round(x)`	The nearest integer (round-off)
`acosh(x)`	$\cosh^{-1}(x)$	`fix(x)`	The nearest integer toward 0
`asinh(x)`	$\sinh^{-1}(x)$	`floor(x)`	The greatest integer $\le x$
`atanh(x)`	$\tanh^{-1}(x)$	`ceil(x)`	The smallest integer $\ge x$
`max`	Maximum and its index	`sign(x)`	1(positive)/0/-1(negative)
`min`	Minimum and its index	`mod(y,x)`	Remainder of y/x
`sum`	Sum	`rem(y,x)`	Remainder of y/x
`prod`	Product	`eval(f)`	Evaluate an expression
`norm`	Norm	`feval(f,a)`	Function evaluation
`sort`	Sort in the ascending order	`polyval`	Value of a polynomial function
`clock`	Present time	`poly`	Polynomial with given roots

Table 1.3 (*continued*)

`find`	Index of element(s)	`roots`	Roots of polynomial
`flops(0)`	Reset the flops count to zero	`tic`	Start a stopwatch timer
`flops`	Cumulative # of floating point operations (unavailable in MATLAB 6.x)	`toc`	Read the stopwatch timer (elapsed time from tic)
`date`	Present date	`magic`	Magic square
Reserved Variables with Special Meaning			
`i,j`	$\sqrt{-1}$	`pi`	π
`eps`	Machine epsilon floating point relative accuracy	`realmax realmin`	Largest/smallest positive number
`break`	Exit while/for loop	`Inf, inf`	Largest number (∞)
`end`	The end of for-loop or if, while, case statement or an array index	`NaN`	Not_a_Number (undetermined)
`nargin`	Number of input arguments	`nargout`	Number of output arguments
`varargin`	Variable input argument list	`varargout`	Variable output argument list

Once we store these functions into the files named ‘f1.m’ and ‘f49.m’ after the function names, respectively, we can call and use them as needed inside another M-file or in the MATLAB Command window.

```
>>f1([0 1]) %several values of a scalar function of a scalar variable
   ans = 1.0000   0.1111
>>f49([0 1]) %a value of a 2-D vector function of a vector variable
   ans = -1.0000  -5.5000
>>feval('f1',[0 1]), feval('f49',[0 1]) %equivalently, yields the same
   ans =  1.0000   0.1111
   ans = -1.0000  -5.5000
```

(Q5) With the function f1(x) defined as a scalar function of a scalar variable, we enter a vector as its input argument to obtain a seemingly vector-valued output. What’s going on?

(A5) It is just a set of function values [f1(x1) f1(x2) ...] obtained at a time for several values [x1 x2...] of x. In expectation of one-shot multi-operation, it is a good practice to put a dot(.) just before the arithmetic operators *(multiplication), /(division), and ^ (power) in the function definition so that the term-by-term (termwise) operation can be done any time.

Note that we can define a simple function not only in an independent M-file, but also inside a program by using the `inline()` command or just in a form of literal expression that can be evaluated by the command `eval()`.

```
>>f1 = inline('1./(1+8*x.^2)','x');
>>f1([0 1]), feval(f1,[0 1])
   ans = 1.0000   0.1111
   ans = 1.0000   0.1111
>>f1 = '1./(1+8*x.^2)'; x = [0 1]; eval(f1)
   ans = 1.0000   0.1111
```

As far as a polynomial function is concerned, it can simply be defined as its coefficient vector arranged in descending order. It may be called to yield its value for certain value(s) of its independent variable by using the command `polyval()`.

```
>>p = [1 0 -3 2]; %polynomial function  p(x) = x^3 - 3x + 2
>>polyval(p,[0 1])
   ans =  2.0000   0.0000
```

The multiplication of two polynomials can be performed by taking the convolution of their coefficient vectors representing the polynomials in MATLAB, since

$$(a_N x^N + \cdots + a_1 x + a_0)(b_N x^N + \cdots + b_1 x + b_0) = c_{2N} x^{2N} + \cdots + c_1 x + c_0$$

where

$$c_k = \sum_{m=\max(0,k-N)}^{\min(k,N)} a_{k-m} b_m \qquad \text{for } k = 2N, 2N-1, \ldots, 1, 0$$

This operation can be done by using the MATLAB built-in command `conv()` as illustrated below.

```
>>a = [1 -1]; b=[1 1 1]; c = conv(a,b)
   c = 1   0   0  -1 %meaning that (x - 1)(x^2 + x + 1) = x^3 + 0 · x^2 + 0 · x - 1
```

But, in case you want to multiply a polynomial by only x^n, you can simply append n zeros to the right end of the polynomial coefficient vector to extend its dimension.

```
>>a = [1 2 3]; c = [a 0 0] %equivalently, c = conv(a,[1 0 0])
   c = 1  2  3  0  0 %meaning that (x^2 + 2x + 3)x^2 = x^4 + 2x^3 + 3x^2 + 0 · x + 0
```

1.1.7 Operations on Vectors and Matrices

We can define a new scalar/vector/matrix or redefine any existing ones in terms of the existent ones or irrespective of them. In the MATLAB Command window, let us define$A$ and B as

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \qquad B = \begin{bmatrix} 3 \\ -2 \\ 1 \end{bmatrix}$$

by typing

```
>>A = [1 2 3;4 5 6], B = [3;-2;1]
```

We can modify them or take a portion of them. For example:

```
>>A = [A;7 8 9]
     A = 1     2      3
         4     5      6
         7     8      9
>>B = [B [1 0 -1]']
    B = 3      1
       -2      0
        1     -1
```

Here, the apostrophe (prime) operator (`'`) takes the complex conjugate transpose and functions virtually as a transpose operator for real-valued matrices. If you want to take just the transpose of a complex-valued matrix, you should put a dot(`.`) before `'`, that is, ‘`.'`’.

When extending an existing matrix or defining another one based on it, the compatibility of dimensions should be observed. For instance, if you try to annex a 4×1 matrix into the 3×1 matrix B, MATLAB will reject it squarely, giving you an error message.

```
>>B = [B ones(4,1)]
  ???All matrices on a row in the bracketed expression must have
     the same number of rows
```

We can modify or refer to a portion of a given matrix.

```
>>A(3,3) = 0
  A = 1    2    3
      4    5    6
      7    8    0
>>A(2:3,1:2) %from 2nd row to 3rd row, from 1st column to 2nd column
  ans = 4    5
        7    8
>>A(2,:) %2nd row, all columns
  ans = 4   5   6
```

The colon (`:`) is used for defining an arithmetic (equal difference) sequence without the bracket `[]` as

```
>>t = 0:0.1:2
```

which makes

```
t = [0.0 0.1 0.2 ... 1.9 2.0]
```

(Q6) What if we omit the increment between the left/right boundary numbers?

(A6) By default, the increment is 1.

```
>>t = 0:2
  t = 0 1 2
```

(Q7) What if the right boundary number is smaller/greater than the left boundary number with a positive/negative increment?

(A7) It yields an empty matrix, which is useless.

```
>>t = 0:-2
  t = Empty matrix: 1-by-0
```

(Q8) If we define just some elements of a vector not fully, but sporadically, will we have a row vector or a column vector and how will it be filled in between?

(A8) We will have a row vector filled with zeros between the defined elements.

```
>>D(2) = 2; D(4) = 3
  D = 0   2   0   3
```

(Q9) How do we make a column vector in the same style?

(A9) We must initialize it as a (zero-filled) row vector, prior to giving it a value.

```
>>D = zeros(4,1); D(2) = 2; D(4) = 3
  D = 0
        2
        0
        3
```

(Q10) What happens if the specified element index of an array exceeds the defined range?

(A10) It is rejected. MATLAB does not accept nonpositive or noninteger indices.

```
>>D(5)
   ??? Index exceeds matrix dimensions.
>>D(0) = 1;
   ??? Index into matrix is negative or zero.
>>D(1.2)
   ??? Subscript indices must either be real positive
       integers ..
```

(Q11) How do we know the size (the numbers of rows/columns) of an already-defined array?

(A11) Use the `length()` and `size()` commands as indicated below.

```
>>length(D)
   ans  = 4
>>[M,N] = size(A)
   M  = 3
   N  = 3
```

MATLAB enables us to handle vector/matrix operations in almost the same way as scalar operations. However, we must make sure of the dimensional compatibility between vectors/matrices, and we must put a dot (`.`) in front of the operator for termwise (element-by-element) operations. The addition of a matrix and a scalar adds the scalar to every element of the matrix. The multiplication of a matrix by a scalar multiplies every element of the matrix by the scalar.

There are several things to know about the matrix division and inversion.

Remark 1.1. Rules of Vector/Matrix Operation

1. For a matrix to be invertible, it must be square and nonsingular; that is, the numbers of its rows and columns must be equal and its determinant must not be zero.
2. The MATLAB command `pinv(A)` provides us with a matrix X of the same dimension as A^T such that $AXA = A$ and $XAX = X$. We can use this command to get the right/left pseudo- (generalized) inverse $A^T[AA^T]^{-1}/[A^TA]^{-1}A^T$ for a matrix A given as its input argument, depending on whether the number (M) of rows is smaller or greater than the number (N) of columns, so long as the matrix is of full rank; that is, rank$(A) = \min(M, N)$[K-1, Section 6.4]. Note that $A^T[AA^T]^{-1}/[A^TA]^{-1}A^T$ is called the right/left inverse because it is multiplied onto the right/left side of A to yield an identity matrix.
3. You should be careful when using the `pinv(A)` command for a rank-deficient matrix, because its output is no longer the right/left inverse, which does not even exist for rank-deficient matrices.
4. The value of a scalar function having an array value as its argument is also an array with the same dimension.

Suppose we have defined vectors a_1, a_2, b_1, b_2 and matrices A_1, A_2, B as follows:

```
>>a1 = [-1 2 3]; a2 = [4 5 2]; b1 = [1 -3]'; b2 = [-2 0];
```

$$a_1 = [-1 \ \ 2 \ \ 3], \quad a_2 = [4 \ \ 5 \ \ 2], \quad b_1 = \begin{bmatrix} 1 \\ -3 \end{bmatrix}, \quad b_2 = [-1 \ \ 2 \ \ 3]$$

```
>>A1 = [a1;a2], A2 = [a1;[b2 1]], B = [b1 b2']
```

$$A_1 = \begin{bmatrix} -1 & 2 & 3 \\ 4 & 5 & 2 \end{bmatrix}, \qquad A_2 = \begin{bmatrix} -1 & 2 & 3 \\ -2 & 0 & 1 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & -2 \\ -3 & 0 \end{bmatrix}$$

The results of various operations on these vectors/matrices are as follows (pay attention to the error message):

```
>>A3 = A1 + A2, A4 = A1 - A2, 1 + A1 %matrix/scalar addition/subtraction
    A3 = -2  4  6    A4 = 0  0  0     ans = 0  3  4
          2  5  3         6  5  1           5  6  3
>>AB = A1*B % AB(m,n) = sum_k A1(m,k)B(k,n) matrix multiplication?
   ??? Error using  ==> *
   Inner matrix dimensions must agree.
>>BA1 = B*A1 % regular matrix multiplication
  BA1 = -9  -8  -1
         3  -6  -9
>>AA = A1.*A2 %termwise multiplication
  AA =  1   4   9
       -8   0   2
>>AB=A1.*B % AB(m,n) = A1(m,n)B(m,n) termwise multiplication
   ??? Error using  ==> .*
   Matrix dimensions must agree.
>>A1_1 = pinv(A1),A1'*(A1*A1')^-1,eye(size(A1,2))/A1 % A1^T[A1A1^T]^-1
  A1_1 =  -0.1914    0.1399   %right inverse of a 2 x 3 matrix A1
           0.0617    0.0947
           0.2284   -0.0165
>>A1*A1_1 %A1/A1 = I implies the validity of A1_1 as the right inverse
   ans = 1.0000    0.0000
         0.0000    1.0000
>>A5 = A1'; % a 3 x 2 matrix
>>A5_1 = pinv(A5),(A5'*A5)^-1*A5',A5\eye(size(A5,1)) % [A5^TA5]^-1A5^T
  A5_1 =  -0.1914   0.0617    0.2284  %left inverse of a 3x2 matrix A5
           0.1399   0.0947   -0.0165
>>A5_1*A5 % = I implies the validity of A5_1 as the left inverse
    ans =  1.0000   -0.0000
          -0.0000    1.0000
>>A1_li = (A1'*A1)^-1*A1' %the left inverse of matrix A1 with M < N?
    Warning: Matrix is close to singular or badly scaled.
              Results may be inaccurate. RCOND  = 9.804831e-018.
    A1_li  = -0.2500    0.2500
              0.2500    0
              0.5000    0.5000
```

(Q12) Does the left inverse of a matrix having rows fewer than columns exist?

(A12) No. There is no $N \times M$ matrix that is premultiplied on the left of an $M \times N$ matrix with $M < N$ to yield a nonsingular matrix, far from an identity matrix. In this context, MATLAB should have rejected the above case on the ground that $[A_1^T A_1]$ is singular and so its inverse does not exist. But, because the round-off errors make a very small number appear to be a zero or make a real zero appear to be a very small number (as will be mentioned in Remark 2.3), it is not easy for MATLAB to tell a near-singularity from a real singularity. That is why MATLAB dares not to declare the singularity case and instead issues just a warning message to remind you to check the validity of the result so that it will not be blamed for a delusion. Therefore, you must be alert for the condition

mentioned in item 2 of Remark 1.1, which says that, in order for the left inverse to exist, the number of rows must not be less than the number of columns.

```
>>A1_li*A1 %No identity matrix, since A1_li isn't the left inverse
  ans = 1.2500    0.7500   -0.2500
       -0.2500    0.5000    0.7500
        1.5000    3.5000    2.5000
>>det(A1'*A1) %A1 is not left-invertible for A1'*A1 is singular
    ans = 0
```

(cf) Let us be nice to MATLAB as it is to us. From the standpoint of promoting mutual understanding between us and MATLAB, we acknowledge that MATLAB tries to show us apparently good results to please us like always, sometimes even pretending not to be obsessed by the demon of 'ill-condition' in order not to make us feel uneasy. How kind MATLAB is! But, we should be always careful not to be spoiled by its benevolence and not to accept the computing results every inch as it is. In this case, even though the matrix `[A1'*A1]` is singular and so not invertible, MATLAB tried to invert it and that's all. MATLAB must have felt something abnormal as can be seen from the ominous warning message prior to the computing result. Who would blame MATLAB for being so thoughtful and loyal to us? We might well be rather touched by its sincerity and smartness.

In the above statements, we see the slash(`/`)/backslash(`\`) operators. These operators are used for right/left division, respectively; `B/A` is the same as `B*inv(A)` and `A\B` is the same as `inv(A)*B` when `A` is invertible and the dimensions of `A` and `B` are compatible. Noting that `B/A` is equivalent to `(A'\B')'`, let us take a close look at the function of the backslash(\) operator.

```
>>X = A1\A1 % an identity matrix?
    X = 1.0000    0         -0.8462
        0         1.0000     1.0769
        0         0          0
```

(Q13) It seems that A1\A1 should have been an identity matrix, but it is not, contrary to our expectation. Why?

(A13) We should know more about the various functions of the backslash(\), which can be seen by typing '`help slash`' into the MATLAB Command window. Let Remark 1.2 answer this question in cooperation with the next case.

```
>>A1*X - A1 %zero if X is the solution to A1*X = A1?
   ans = 1.0e-015 * 0         0              0
                    0         0        -0.4441
```

Remark 1.2. The Function of Backslash (\) Operator. Overall, for the command '`A\B`', MATLAB finds a solution to the equation `A*X = B`. Let us denote the row/column dimension of the matrix `A` by `M` and `N`.

1. If matrix `A` is square and upper/lower-triangular in the sense that all of its elements below/above the diagonal are zero, then MATLAB finds the solution by applying backward/forward substitution method (Section 2.2.1).

2. If matrix `A` is square, symmetric (Hermitian), and positive definite, then MATLAB finds the solution by using Cholesky factorization (Section 2.4.2).
3. If matrix `A` is square and has no special feature, then MATLAB finds the solution by using LU decomposition (Section 2.4.1).
4. If matrix `A` is rectangular, then MATLAB finds a solution by using QR factorization (Section 2.4.2). In case `A` is rectangular and of full rank with rank(`A`) = min(`M,N`), it will be the LS (least-squares) solution [Eq. (2.1.10)] for `M > N` (overdetermined case) and one of the many solutions that is not always the same as the minimum-norm solution [Eq. (2.1.7)] for `M < N` (underdetermined case). But for the case when `A` is rectangular and has rank deficiency, what MATLAB gives us may be useless. Therefore, you must pay attention to the warning message about rank deficiency, which might tell you not to count on the dead-end solution made by the backslash (\) operator. To find an alternative in the case of rank deficiency, you had better resort to singular value decomposition (SVD). See Problem 2.8 for details.

For the moment, let us continue to try more operations on matrices.

```
>>A1./A2 %termwise right division
  ans =  1      1      1
        -2     Inf     2
>>A1.\A2 %termwise left division
  ans =  1      1      1
        -0.5    0      0.5
>>format rat, B^-1 %represent the numbers (of B^{-1}) in fractional form
  ans =  0    -1/3
        -1/2  -1/6
>>inv(B) %inverse matrix, equivalently
  ans =  0      -1/3
        -1/2    -1/6
>>B.^-1 %termwise inversion(reciprocal of each element)
  ans =  1      -1/2
        -1/3     Inf
>>B^2 %square of B, i.e., B^2 = B*B
  ans =  7      -2
        -3       6
>>B.^2 %termwise square(square of each element)
  ans =  1(b_11^2)    4(b_12^2)
         9(b_21^2)    0(b_22^2)
>>2.^B %2 to the power of each number in B
  ans =   2 (2^b11)   1/4(2^b12)
         1/8(2^b21)    1 (2^b22)
>>A1.^A2 %element of A1 to the power of each element in A2
  ans =  -1 (A1(1,1)^A2(1,1))    4(A1(1,2)^A2(1,2))    27(A1(1,3)^A2(1,3))
        1/16(A1(2,1)^A2(2,1))    1(A1(2,2)^A2(2,2))     2(A1(2,3)^A2(2,3))
>>format short, exp(B) %elements of e^B  with 4 digits below the dp
  ans = 2.7183(e^b11)    0.1353(e^b12)
        0.0498(e^b21)    1.0000(e^b22)
```

There are more useful MATLAB commands worthwhile to learn by heart.

Remark 1.3. More Useful Commands for Vector/Matrix Operations

1. We can use the commands `zeros()`, `ones()`, and `eye()` to construct a matrix of specified size or the same size as an existing matrix which has only zeros, only ones, or only ones/zeros on/off its diagonal.

```
>>Z = zeros(2,3) %or zeros(size(A1)) yielding a 2 x 3 zero matrix
  Z = 0    0    0
      0    0    0
>>E = ones(size(B)) %or ones(3,2) yielding a 3 x 2 one matrix
  E = 1    1
      1    1
      1    1
>>I = eye(2) %yielding a 2 x 2 identity matrix
  I = 1    0
      0    1
```

2. We can use the `diag()` command to make a column vector composed of the diagonal elements of a matrix or to make a diagonal matrix with on-diagonal elements taken from a vector given as the input argument.

```
>>A1, diag(A1) %column vector consisting of diagonal elements
    A1 =  -1    2    3
           4    5    2
    ans = -1
           5
```

3. We can use the commands `sum()/prod()` to get the sum/product of elements in a vector or a matrix, columnwisely first (along the first non-singleton dimension).

```
>>sa1 = sum(a1) %sum of all the elements in vector a1
  sa1 = 4 %∑ a1(n) = − 1 + 2 + 3 = 4
>>sA1 = sum(A1) %sum of all the elements in each column of matrix A1
  sA1 = 3   7   5 %sA1(n) = ∑_{m=1}^{M} A1(m,n) = [− 1 + 4   2 + 5   3 + 2]
>>SA1 = sum(sum(A1)) %sum of all elements in matrix A1
  SA1 = 15  %SA1 = ∑_{n=1}^{N} ∑_{m=1}^{M} A1(m,n) = 3 + 7 + 5 = 15
>>pa1 = prod(a1) %product of all the elements in vector a1
  pa1 = 4 %∏ a1(n) = ( − 1) × 2 × 3 = − 6
>>pA1=product(A1) %product of all the elements in each column of matrix A1
  pA1 = -4   10   6 %pA1(n) = ∏_{m=1}^{M} A1(m,n) = [−1 × 4   2 × 5   3 × 2]
>>PA1 = product(product(A1)) %product of all the elements of matrix A1
  PA1 = -240 %PA1 = ∏_{n=1}^{N} ∏_{m=1}^{M} A1(m,n) = ( − 4) × 10 × 6 = − 240
```

4. We can use the commands `max()/min()` to find the first maximum/minimum number and its index in a vector or in a matrix given as the input argument.

```
>>[aM,iM] = max(a2)
  aM = 5, iM = 2 %means that the max. element of vector a2 is a2(2) = 5
>>[AM,IM] = max(A1)
  AM = 4  5  3
  IM = 2  2  1
  %means that the max. elements of each column of A1 are
     A1(2,1) = 4, A1(2,2) = 5, A1(1,3) = 3
```

```
>>[AMx,J] = max(AM)
  AMx = 5, J = 2
  %implies that the max. element of A1 is A1(IM(J),J) = A1(2,2) = 5
```

5. We can use the commands `rot90()/fliplr()/flipud()` to rotate a matrix by an integer multiple of 90° and to flip it left-right/up-down.

```
>>A1, A3 = rot90(A1), A4 = rot90(A1,-2)
  A1 = -1     2     3
        4     5     2
  A3 =  3     2       %90° rotation
        2     5
       -1     4
  A4 =  2     5     4 %90°x(-2) rotation
        3     2    -1
>>A5 = fliplr(A1) %flip left-right
  A5 =  3     2    -1
        2     5     4
>>A6 = flipud(A1) %flip up-down
  A6 =  4    5    2
       -1    2    3
```

6. We can use the `reshape()` command to change the row-column size of a matrix with its elements preserved (columnwisely first).

```
>>A7 = reshape(A1,3,2)
  A7 = -1     5
        4     3
        2     2
>>A8 = reshape(A1,6,1), A8 = A1(:) %makes supercolumn vector
  A8 = -1
        4
        2
        5
        3
        2
```

1.1.8 Random Number Generators

MATLAB has the built-in functions, `rand()/randn()`, to generate random numbers having uniform/normal (Gaussian) distributions, respectively ([K-1], Chapter 22).

`rand(M,N)`: generates an `M x N` matrix consisting of uniformly distributed random numbers `randn(M,N)`: generates an `M x N` matrix consisting of normally distributed random numbers

1. Random Number Having Uniform Distribution

The numbers in a matrix generated by the MATLAB function `rand(M,N)` have uniform probability distribution over the interval [0,1], as described by U(0,1). The random number x generated by `rand()` has the probability density function

$$f_X(x) = u_s(x) - u_s(x-1) \quad (u_s(x) = \begin{cases} 1 & \forall x \geq 0 \\ 0 & \forall x < 0 \end{cases} : \text{the unit step function}) \tag{1.1.1}$$

whose value is 1 over [0,1] and 0 elsewhere. The average of this standard uniform number x is

$$m_X = \int_{-\infty}^{\infty} x f_X(x)dx = \int_0^1 x\,dx = \left.\frac{x^2}{2}\right|_0^1 = \frac{1}{2} \tag{1.1.2}$$

and its variance or deviation is

$$\sigma_X^2 = \int_{-\infty}^{\infty} (x - m_X)^2 f_X(x)dx = \int_0^1 (x - \frac{1}{2})^2 dx = \frac{1}{3}\left.(x - \frac{1}{2})^3\right|_0^1 = \frac{1}{12} \tag{1.1.3}$$

If you want another random number y with uniform distribution U(a, b), transform the standard uniform number x as follows:

$$y = (b - a)x + a \tag{1.1.4}$$

For practice, we make a vector consisting of 1000 standard uniform numbers, transform it to make a vector of numbers with uniform distribution U(-1, $+1$), and then draw the histograms showing the shape of the distribution for the two uniform number vectors (Fig. 1.7a,b).

```
>>u_noise = rand(1000,1) %a 1000x1 noise vector with U(0,1)
>>subplot(221), hist(u_noise,20) %histogram having 20 divisions
```

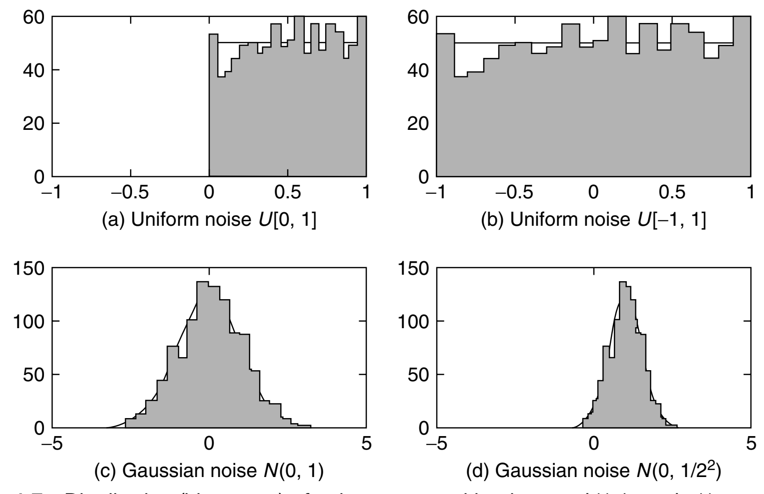


Figure 1.7 Distribution (histogram) of noise generated by the `rand()`/`randn()` command.

```
>>u_noise1 = 2*u_noise-1 %a 1000x1 noise vector with U(-1,1)
>>subplot(222), hist(u_noise1,20) %histogram
```

2. Random Number with Normal (Gaussian) Distribution

The numbers in a matrix generated by the MATLAB function `randn(M,N)` have normal (Gaussian) distribution with average $m = 0$ and variance $\sigma^2 = 1$, as described by $N(0,1)$. The random number x generated by `rand()` has the probability density function

$$f_X(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \tag{1.1.5}$$

If you want another Gaussian number y with a general normal distribution $N(m, \sigma^2)$, transform the standard Gaussian number x as follows:

$$y = \sigma\ x + m \tag{1.1.6}$$

The probability density function of the new Gaussian number generated by this transformation is obtained by substituting $x = (y - m)/\sigma$ into Eq. (1.1.5) and dividing the result by the scale factor σ (which can be seen in $dx = dy/\sigma$) so that the integral of the density function over the whole interval $(-\infty, +\infty)$ amounts to 1.

$$f_Y(y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(y-m)^2/2\sigma^2} \tag{1.1.7}$$

For practice, we make a vector consisting of 1000 standard Gaussian numbers, transform it to make a vector of numbers having normal distribution $N(1,1/4)$, with mean $m = 1$ and variance $\sigma^2 = 1/4$, and then draw the histograms for the two Gaussian number vectors (Fig. 1.7c,d).

```
>>g_noise = randn(1000,1) %a 1000x1 noise vector with N(0,1)
>>subplot(223), hist(g_noise,20) %histogram having 20 divisions
>>g_noise1 = g_noise/2+1 %a 1000x1 noise vector with N(1,1/4)
>>subplot(224), hist(g_noise1,20) %histogram
```

1.1.9 Flow Control

1. `if-end` ***and*** `switch-case-end` ***Statements***

An `if-end` block basically consists of an *`if`* statement, a sequel part, and an *`end`* statement categorizing the block. An *`if`* statement, having a condition usually based on the relational/logical operator (Table 1.4), is used to control the program flow—that is, to adjust the order in which statements are executed according to whether or not the condition is met, mostly depending on unpredictable situations. The sequel part consisting of one or more statements may contain *`else`* or *`elseif`* statements, possibly in a nested structure containing another *`if`* statement inside it.

The `switch-case-end` block might replace a multiple `if-elseif-..-end` statement in a neat manner.

Table 1.4 Relational Operators and Logical Operators

Relational operator	Remark	Relational operator	Remark	Logical operator	Remark
<	less than	>	greater than	&	and
<=	less than or equal to	>=	greater than or equal to	|	or
==	equal	~=	not equal($\neq$)	~	not

Let us see the following examples:

Example 1. A Simple `if-else-end` *Block*

```
%nm119_1: example of if-end block
t = 0;
if t > 0
   sgnt = 1;
 else
   sgnt = -1;
end
```

Example 2. A Simple `if-elseif-end` *Block*

```
%nm119_2: example of if-elseif-end block
if t > 0
 sgnt = 1
 elseif t < 0
   sgnt = -1
end
```

Example 3. An `if-elseif-else-end` *Block*

```
%nm119_3: example of if-elseif-else-end block
if t > 0, sgnt = 1
 elseif t<0, sgnt = -1
 else   sgnt = 0
end
```

Example 4. An `if-elseif-elseif-...-else-end` *Block*

```
%nm119_4: example of if-elseif-elseif-else-end block
point = 85;
if point >= 90, grade = 'A'
  elseif point >= 80, grade = 'B'
  elseif point >= 70, grade = 'C'
  elseif point >= 60, grade = 'D'
  else                grade = 'F'
end
```

Example 5. A `switch-case-end` *Block*

```
%nm119_5: example of switch-case-end block
point = 85;
switch floor(point/10) %floor(x): integer less than or equal to x
  case  9, grade = 'A'
  case  8, grade = 'B'
  case  7, grade = 'C'
  case  6, grade = 'D'
  otherwise grade = 'F'
end
```

2. `for index = i_0:increment:i_last-end` Loop

A `for` loop makes a block of statements executed repeatedly for a specified number of times, with its loop index increasing from `i_0` to a number not greater than `i_last` by a specified step (`increment`) or by 1 if not specified. The loop iteration normally ends when the loop index reaches `i_last`, but it can be stopped by a `break` statement inside the `for` loop. The `for` loop with a positive/negative increment will never be iterated if the last value (`i_last`) of the index is smaller/greater than the starting value (`i_0`).

Example 6. A `for` *Loop*

```
%nm119_6: example of for loop
point = [76 85 91 65 87];
for n = 1:length(point)
   if point(n) >= 80,  pf(n,:) = 'pass';
     elseif point(n) >= 0, pf(n,:) = 'fail';
     else  %if point(n)< 0
     pf(n,:) = '????';
     fprintf('\n\a Something wrong with the data??\n');
     break;
   end
end
pf
```

3. `while` Loop

A `while` loop will be iterated as long as its predefined condition is satisfied and a `break` statement is not encountered inside the loop.

Example 7. A `while` *Loop*

```
%nm119_7: example of while loop
r = 1;
while r < 10
  r = input('\nType radius (or nonpositive number to stop):');
  if r <= 0, break, end %isempty(r)|  r  <=  0, break, end
  v = 4/3*pi*r*r*r;
  fprintf('The volume of a sphere with radius %3.1f  = %8.2f\n',r,v);
end
```

Example 8. `while` *Loops to Find the Minimum/Maximum Positive Numbers*

The following program "nm119_8.m" contains three `while` loops. In the first one, `x = 1` continues to be divided by 2 until just before reaching zero, and it will hopefully end up with the smallest positive number that can be represented in MATLAB. In the second one, `x = 1` continues to be multiplied by 2 until just before reaching `inf` (the infinity defined in MATLAB), and seemingly it will get the largest positive number (`x_max0`) that can be represented in MATLAB. But, while this number reaches or may exceed `inf` if multiplied by 2 once more, it still is not the largest number in MATLAB (slightly less than `inf`) that we want to find. How about multiplying `x_max0` by $(2 - 1/2^n)$? In the third `while` loop, the temporary variable `tmp` starting with the initial value of 1 continues to be divided by 2 until just before `x_max0*(2-tmp)` reaches `inf,` and apparently it will end up with the largest positive number (`x_max`) that can be represented in MATLAB.

```
%nm119_8: example of while loops
x = 1; k1 = 0;
while x/2 > 0
   x = x/2; k1 = k1 + 1;
end
k1, x_min = x;
fprintf('x_min is %20.18e\n',x_min)

x = 1; k2 = 0;
while 2*x < inf
   x = x*2; k2 = k2+1;
end
k2, x_max0 = x;

tmp = 1; k3 = 0;
while x_max0*(2-tmp/2) < inf
  tmp = tmp/2; k3 = k3+1;
end
k3, x_max = x_max0*(2-tmp);
fprintf('x_max is %20.18e\n',x_max)

format long e
x_min,-x_min,x_max,-x_max
format hex
x_min,-x_min,x_max,-x_max
format short
```

1.2 COMPUTER ERRORS VERSUS HUMAN MISTAKES

Digital systems like calculators and computers hardly make a mistake, since they follow the programmed order faithfully. Nonetheless, we often encounter some numerical errors in the computing results made by digital systems, mostly coming from representing the numbers in finite bits, which is an intrinsic limitation of digital world. If you let the computer compute something without considering what is called the finite-word-length effect, you might come across a weird answer. In

that case, it is not the computer, but yourself as the user or the programmer, who is to blame for the wrong result. In this context, we should always be careful not to let the computer produce a farfetched output. In this section we will see how the computer represents and stores the numbers. Then we think about the cause and the propagation effect of computational error in order not to be deceived by unintentional mistakes of the computer and, it is hoped, to be able to take some measures against them.

1.2.1 IEEE 64-bit Floating-Point Number Representation

MATLAB uses the IEEE 64-bit floating-point number system to represent all numbers. It has a word structure consisting of the sign bit, the exponent field, and the mantissa field as follows:

63	62 ... 52	51 ... 0
S	Exponent	Mantissa

Each of these fields expresses S, E, and M of a number f in the way described below.

- Sign bit

$$S = b_{63} = \begin{cases} 0 & \text{for positive numbers} \\ 1 & \text{for negative numbers} \end{cases}$$

- Exponent field ($b_{62}b_{61}b_{60}\cdots b_{52}$): adopting the excess 1023 code

$$\begin{aligned} E &= \text{Exp} - 1023 = \{0, 1, \ldots, 2^{11} - 1 = 2047\} - 1023 \\ &= \{-1023, -1022, \ldots, +1023, +1024\} \\ &= \begin{cases} -1023 + 1 & \text{for } |f| < 2^{-1022}(\text{Exp} = 00000000000) \\ -1022 \sim +1023 & \text{for } 2^{-1022} \le |f| < 2^{1024}(\text{normalized ranges}) \\ +1024 & \text{for } \pm\infty \end{cases} \end{aligned}$$

- Mantissa field ($b_{51}b_{50}\ldots b_1b_0$):

In the un-normalized range where the numbers are so small that they can be represented only with the value of hidden bit 0, the number represented by the mantissa is

$$M = 0.b_{51}b_{50}\cdots b_1b_0 = [b_{51}b_{50}\cdots b_1b_0] \times 2^{-52} \tag{1.2.1}$$

You might think that the value of the hidden bit is added to the exponent, instead of to the mantissa.

In the normalized range, the number represented by the mantissa together with the value of hidden bit $b_h = 1$ is

$$\begin{aligned} M &= 1.b_{51}b_{50}\cdots b_1b_0 = 1 + [b_{51}b_{50}\cdots b_1b_0] \times 2^{-52} \\ &= 1 + b_{51} \times 2^{-1} + b_{50} \times 2^{-2} + \cdots + b_1 \times 2^{-51} + b_0 \times 2^{-52} \end{aligned}$$

$$= \{1, 1 + 2^{-52}, 1 + 2 \times 2^{-52}, \ldots, 1 + (2^{52} - 1) \times 2^{-52}\}$$
$$= \{1, 1 + 2^{-52}, 1 + 2 \times 2^{-52}, \ldots, (2 - 2^{-52})\}$$
$$= \{1, 1 + \Delta, 1 + 2\Delta, \ldots, 1 + (2^{52} - 1)\Delta = 2 - \Delta\}\ (\Delta = 2^{-52}) \quad (1.2.2)$$

The set of numbers S, E, and M, each represented by the sign bit S, the exponent field Exp and the mantissa field M, represents a number as a whole

$$f = \pm M \cdot 2^E \quad (1.2.3)$$

We classify the range of numbers depending on the value (E) of the exponent and denote it as

$$R_E = [2^E, 2^{E+1}) \qquad \text{with } -1022 \leq E \leq +1023 \quad (1.2.4)$$

In each range, the least unit—that is, the value of LSB (least significant bit) or the difference between two consecutive numbers represented by the mantissa of 52 bits—is

$$\Delta_E = \Delta \times 2^E = 2^{-52} \times 2^E = 2^{E-52} \quad (1.2.5)$$

Let us take a closer look at the bitwise representation of numbers belonging to each range.

0. 0(zero)

63	62 … 52	51 … 0
S	000 . . 0000	0000 0000 . . . 0000 0000

1. Un-normalized Range (with the value of hidden bit $b_h = 0$)

$R_{-1023} = [2^{-1074}, 2^{-1022})$ with Exp $= 0$, $E =$ Exp $- 1023 \underline{+ 1} = -1022$

S	000 . . . 0000	0000 0000 . . . . 0000 0001	$(0 + 2^{-52}) \times 2^E = (0 + 2^{-52}) \times 2^{-1022}$

. . . .

S	000 . . . 0000	1111 1111 . . . . 1111 1111	$\{(0 + (2^{52} - 1)2^{-52}) = (1 - 2^{-52})\} \times 2^{-1022}$

Value of LSB: $\Delta_{-1023} = \Delta_{-1022} = 2^{-1022-52} = 2^{-1074}$

2. The Smallest Normalized Range (with the value of hidden bit $b_h = 1$)

$R_{-1022} = [2^{-1022}, 2^{-1021})$ with Exp $= 1$, $E =$ Exp $- 1023 = -1022$

S	000 . . . 0001	0000 0000 . . . . 0000 0000	$(1 + 0) \times 2^E = (1 + 0) \times 2^{-1022}$
S	000 . . . 0001	0000 0000 . . . . 0000 0001	$(1 + 2^{-52}) \times 2^{-1022}$

. . . .

S	000 . . . 0001	1111 1111 . . . . 1111 1111	$\{(1 + (2^{52} - 1)\, 2^{-52}) = (2 - 2^{-52})\} \times 2^{-1022}$

Value of LSB: $\Delta_{-1022} = 2^{-1022-52} = 2^{-1074}$

3. Basic Normalized Range (with the value of hidden bit $b_h = 1$)

$R_0 = [2^0, 2^1)$ with Exp $= 2^{10} - 1 = 1023$, $E = \text{Exp} - 1023 = 0$

S ¦ 011 . . . 1111 ¦ 0000 0000 . . . . 0000 0000	$(1+0) \times 2^E = (1+0) \times 2^0 = 1$
S ¦ 011 . . . 1111 ¦ 0000 0000 . . . . 0000 0001	$(1+2^{-52}) \times 2^0$
. . . .	
S ¦ 011 . . . 1111 ¦ 1111 1111 . . . . 1111 1111	$\{(1+(2^{52}-1)\,2^{-52}) = (2-2^{-52})\} \times 2^0$

Value of LSB: $\Delta_0 = 2^{-52}$

4. The Largest Normalized Range (with the value of hidden bit $b_h = 1$)

$R_{1024} = [2^{1023}, 2^{1024})$ with Exp $= 2^{11} - 2 = 2046$, $E = \text{Exp} - 1023 = 1023$

S ¦ 111 . . . 1110 ¦ 0000 0000 . . . . 0000 0000	$(1+0) \times 2^E = (1+0) \times 2^{1023}$
S ¦ 111 . . . 1110 ¦ 0000 0000 . . . . 0000 0001	$(1+2^{-52}) \times 2^{1023}$
. . . .	
S ¦ 111 . . . 1110 ¦ 1111 1111 . . . . 1111 1111	$\{(1+(2^{52}-1)\,2^{-52}) = (2-2^{-52})\} \times 2^{1023}$

Value of LSB: $\Delta_{-1022} = 2^{-1022-52} = 2^{-1074}$

5. $\pm\infty$(inf) Exp $= 2^{11} - 1 = 2047$, $E = \text{Exp} - 1023 = 1024$ (meaningless)

0 ¦ 111 . . . 1111 ¦ 0000 0000 . . . . 0000 0000	$+\infty \neq (1+0) \times 2^E = (1+0) \times 2^{1024}$
1 ¦ 111 . . . 1111 ¦ 0000 0000 . . . . 0000 0000	$-\infty \neq -(1+0) \times 2^E = -(1+0) \times 2^{1024}$
S ¦ 111 . . . 1111 ¦ 0000 0000 . . . . 0000 0001	invalid (not used)
. . . .	
S ¦ 111 . . . 111 ¦ 1111 1111 . . . . 1111 1111	invalid (not used)

From what has been mentioned earlier, we know that the minimum and maximum positive numbers are, respectively,

$$f_{\min} = (0 + 2^{-52}) \times 2^{-1022} = 2^{-1074} = 4.9406564584124654 \times 10^{-324}$$
$$f_{\max} = (2 - 2^{-52}) \times 2^{1023} = 1.7976931348623157 \times 10^{308}$$

This can be checked by running the program "`nm119_8.m`" in Section 1.1.9.

Now, in order to gain some idea about the arithmetic computational mechanism, let's see how the addition of two numbers, 3 and 14, represented in the IEEE 64-bit floating number system, is performed.

```
Digital-to-Binary Conversion→ Normalization→ 64-bit Representation

 3_10 = 11_2    = 1.1_2  × 2^1 = [1].1_2  × 2^(1024−1023)
14_10 = 1110_2  = 1.11_2 × 2^3 = [1].11_2 × 2^(1026−1023)

                        hidden bit  64-bit representation
    3_10 = 0   1024_10   [1].10000... ...0 \
+) 14_10 = 0   1026_10   [1].11000... ...0  ) alignment
    3_10 = 0   1026_10   [0].01100... ...0 /
+) 14_10 = 0   1026_10   [1].11000... ...0
                        carry bit = 1
normalize  0   1026_10   10.00100... ...0
           0   1027_10   [1].00010... ...0 = 1.0001_2 × 10^(1027−1023) = 10001_2 = 1 × 2^4 + 1 × 2^0 = 17_10
                                            Binary-to-Decimal Conversion

2)3  ... 1          2)14 ... 0
   1                2) 7 ... 1
 3_10 = 11_2        2) 3 ... 1
                        11
                    14_10 = 1110_2
```

In the process of adding the two numbers, an alignment is made so that the two exponents in their 64-bit representations equal each other; and it will kick out the part smaller by more than 52 bits, causing some numerical error. For example, adding 2^{-23} to 2^{30} does not make any difference, while adding 2^{-22} to 2^{30} does, as we can see by typing the following statements into the MATLAB Command window.

```
>>x = 2^30; x + 2^-22 == x, x + 2^-23 == x
    ans = 0(false)       ans = 1(true)
```

(cf) Each range has a different minimum unit (LSB value) described by Eq. (1.2.5). It implies that the numbers are uniformly distributed within each range. The closer the range is to 0, the denser the numbers in the range are. Such a number representation makes the absolute quantization error large/small for large/small numbers, decreasing the possibility of large relative quantization error.

1.2.2 Various Kinds of Computing Errors

There are various kinds of errors that we encounter when using a computer for computation.

- *Truncation Error*: Caused by adding up to a finite number of terms, while we should add infinitely many terms to get the exact answer in theory.
- *Round-off Error*: Caused by representing/storing numeric data in finite bits.
- *Overflow/Underflow*: Caused by too large or too small numbers to be represented/stored properly in finite bits—more specifically, the numbers having absolute values larger/smaller than the maximum ($f_{\max}$)/minimum($f_{\min}$) number that can be represented in MATLAB.
- *Negligible Addition*: Caused by adding two numbers of magnitudes differing by over 52 bits, as can be seen in the last section.
- *Loss of Significance*: Caused by a "bad subtraction," which means a subtraction of a number from another one that is almost equal in value.
- *Error Magnification*: Caused and magnified/propagated by multiplying/dividing a number containing a small error by a large/small number.
- Errors depending on the numerical algorithms, step size, and so on.

Although we cannot be free from these kinds of inevitable errors in some degree, it is not computers, but instead human beings, who must be responsible for the computing errors. While our computer may insist on its innocence for an unintended lie, we programmers and users cannot escape from the responsibility of taking measures against the errors and would have to pay for being careless enough to be deceived by a machine. We should, therefore, try to decrease the magnitudes of errors and to minimize their impact on the final results. In order to do so, we must know the sources of computing errors and also grasp the computational properties of numerical algorithms.

For instance, consider the following two formulas:

$$f_1(x) = \sqrt{x}(\sqrt{x+1} - \sqrt{x}), \qquad f_2(x) = \frac{\sqrt{x}}{\sqrt{x+1} + \sqrt{x}} \tag{1.2.6}$$

These are theoretically equivalent, hence we expect them to give exactly the same value. However, running the MATLAB program "nm122.m" to compute the values of the two formulas, we see a surprising result that, as x increases, the step of $f_1(x)$ incoherently moves hither and thither, while $f_2(x)$ approaches 1/2 at a steady pace. We might feel betrayed by the computer and have a doubt about its reliability. Why does such a flustering thing happen with $f_1(x)$? It is because the number of significant bits abruptly decreases when the subtraction $(\sqrt{x+1} - \sqrt{x})$ is performed for large values of x, which is called 'loss of significance'. In order to take a close look at this phenomenon, let $x = 10^{15}$. Then we have

$$\sqrt{x+1} = 3.162277660168381 \times 10^7 = 31622776.60168381$$
$$\sqrt{x} = 3.162277660168379 \times 10^7 = 31622776.60168379$$

These two numbers have 52 significant bits, or equivalently 16 significant digits $(2^{52} \approx 10^{52\times 3/10} \approx 10^{15})$ so that their significant digits range from 10^8 to 10^{-8}. Accordingly, the least significant digit of their sum and difference is also the eighth digit after the decimal point (10^{-8}).

$$\sqrt{x+1} + \sqrt{x} = 63245553.20336761$$
$$\sqrt{x+1} - \sqrt{x} = 0.00000001862645149230957 \approx 0.00000002$$

Note that the number of significant digits of the difference decreased to 1 from 16. Could you imagine that a single subtraction may kill most of the significant digits? This is the very 'loss of significance', which is often called 'catastrophic cancellation'.

```
%nm122
clear
f1 = inline('sqrt(x)*(sqrt(x + 1) - sqrt(x))','x');
f2 = inline('sqrt(x)./(sqrt(x + 1) + sqrt(x))','x');
x = 1;
format long e
for k = 1:15
  fprintf('At x=%15.0f, f1(x)=%20.18f, f2(x) = %20.18f', x,f1(x),f2(x));
  x = 10*x;
end
sx1 = sqrt(x+1);  sx = sqrt(x); d = sx1 - sx;  s = sx1 + sx;
fprintf('sqrt(x+1) = %25.13f, sqrt(x) = %25.13f ',sx1,sx);
fprintf('  diff = %25.23f, sum = %25.23f ',d,s);
```

```
>> nm122
At x=               1, f1(x)=0.41421356237309515O, f2(x)=0.414213562373095090
At x=              10, f1(x)=0.488088481701514750, f2(x)=0.488088481701515480
At x=             100, f1(x)=0.498756211208899460, f2(x)=0.498756211208902730
At x=            1000, f1(x)=0.499875062461021870, f2(x)=0.499875062460964860
At x=           10000, f1(x)=0.499987500624854420, f2(x)=0.499987500624960890
At x=          100000, f1(x)=0.499998750005928860, f2(x)=0.499998750006249940
At x=         1000000, f1(x)=0.499999875046341910, f2(x)=0.499999875000062490
At x=        10000000, f1(x)=0.499999987401150920, f2(x)=0.499999987500000580
At x=       100000000, f1(x)=0.500000005558831620, f2(x)=0.499999998749999950
At x=      1000000000, f1(x)=0.500000077997506340, f2(x)=0.499999999874999990
At x=     10000000000, f1(x)=0.499999441672116520, f2(x)=0.499999999987500050
At x=    100000000000, f1(x)=0.500004449631168080, f2(x)=0.499999999998750000
At x=   1000000000000, f1(x)=0.500003807246685030, f2(x)=0.499999999999874990
At x=  10000000000000, f1(x)=0.499194546973835970, f2(x)=0.499999999999987510
At x= 100000000000000, f1(x)=0.502914190292358400, f2(x)=0.499999999999998720
sqrt(x+1) = 31622776.6016838100000, sqrt(x) = 31622776.6016837920000
  diff=0.0000000186264514923O957, sum=63245553.2033676060000000000000
```

1.2.3 Absolute/Relative Computing Errors

The absolute/relative error of an approximate value x to the true value X of a real-valued variable is defined as follows:

$$\varepsilon_x = X(\text{true value}) - x(\text{approximate value}) \tag{1.2.7}$$

$$\rho_x = \frac{\varepsilon_x}{X} = \frac{X - x}{X} \tag{1.2.8}$$

If the least significant digit (LSD) is the dth digit after the decimal point, then the magnitude of the absolute error is not greater than half the value of LSD.

$$|\varepsilon_x| = |X - x| \le \tfrac{1}{2}10^{-d} \tag{1.2.9}$$

If the number of significant digits is s, then the magnitude of the relative error is not greater than half the relative value of LSD over MSD (most significant digit).

$$|\rho_x| = \frac{|\varepsilon_x|}{|X|} = \frac{|X - x|}{|X|} \le \frac{1}{2}10^{-s} \tag{1.2.10}$$

1.2.4 Error Propagation

In this section we will see how the errors of two numbers, x and y, are propagated with the four arithmetic operations. Error propagation means that the errors in the input numbers of a process or an operation cause the errors in the output numbers.

Let their absolute errors be ε_x and ε_y, respectively. Then the magnitudes of the absolute/relative errors in the sum and difference are

$$\varepsilon_{x\pm y} = (X \pm Y) - (x \pm y) = (X - x) \pm (Y - y) = \varepsilon_x \pm \varepsilon_y$$

$$|\varepsilon_{x\pm y}| \le |\varepsilon_x| + |\varepsilon_y| \tag{1.2.11}$$

$$|\rho_{x\pm y}| = \frac{|\varepsilon_{x\pm y}|}{|X \pm Y|} \le \frac{|X||\varepsilon_x/X| + |Y||\varepsilon_y/Y|}{|X \pm Y|} = \frac{|X||\rho_x| + |Y||\rho_y|}{|X \pm Y|} \tag{1.2.12}$$

From this, we can see why the relative error is magnified to cause the "loss of significance" in the case of subtraction when the two numbers X and Y are almost equal so that $|X - Y| \approx 0$.

The magnitudes of the absolute and relative errors in the multiplication/division are

$$|\varepsilon_{xy}| = |XY - xy| = |XY - (X + \varepsilon_x)(Y + \varepsilon_y)| \approx |X\varepsilon_y \pm Y\varepsilon_x|$$

$$|\varepsilon_{xy}| \leq |X||\varepsilon_y| + |Y||\varepsilon_x| \tag{1.2.13}$$

$$|\rho_{xy}| = \frac{|\varepsilon_{xy}|}{|XY|} \leq \frac{|\varepsilon_y|}{|Y|} + \frac{|\varepsilon_x|}{|X|} = |\rho_x| + |\rho_y| \tag{1.2.14}$$

$$|\varepsilon_{x/y}| = \left|\frac{X}{Y} - \frac{x}{y}\right| = \left|\frac{X}{Y} - \frac{X + \varepsilon_x}{Y + \varepsilon_y}\right| \approx \frac{|X\varepsilon_y - Y\varepsilon_x|}{Y^2}$$

$$|\varepsilon_{x/y}| \leq \frac{|X||\varepsilon_y| + |Y||\varepsilon_x|}{Y^2} \tag{1.2.15}$$

$$|\rho_{x/y}| = \frac{|\varepsilon_{x/y}|}{|X/Y|} \leq \frac{|\varepsilon_x|}{|X|} + \frac{|\varepsilon_y|}{|Y|} = |\rho_x| + |\rho_y| \tag{1.2.16}$$

This implies that, in the worst case, the relative error in multiplication/division may be as large as the sum of the relative errors of the two numbers.

1.2.5 Tips for Avoiding Large Errors

In this section we will look over several tips to reduce the chance of large errors occurring in calculations.

First, in order to decrease the magnitude of round-off errors and to lower the possibility of overflow/underflow errors, make the intermediate result as close to 1 as possible in consecutive multiplication/division processes. According to this rule, when computing xy/z, we program the formula as

- $(xy)/z$ when x and y in the multiplication are very different in magnitude,
- $x(y/z)$ when y and z in the division are close in magnitude, and
- $(x/z)y$ when x and z in the division are close in magnitude.

For instance, when computing y^n/e^{nx} with $x \succ 1$ and $y \succ 1$, we would program it as $(y/e^x)^n$ rather than as y^n/e^{nx}, so that overflow/underflow can be avoided. You may verify this by running the following MATLAB program "nm125_1.m".

```
%nm125_1:
x = 36; y = 1e16;
for n = [-20 -19 19 20]
  fprintf('y^%2d/e^%2dx = %25.15e\n',n,n,y^n/exp(n*x));
  fprintf('(y/e^x)^%2d = %25.15e\n',n,(y/exp(x))^n);
end
```

```
>>nm125_1
  y^-20/e^-20x = 0.000000000000000e+000
  (y/e^x)^-20 =  4.920700930263814e-008
  y^-19/e^-19x = 1.141367814854768e-007
  (y/e^x)^-19 =  1.141367814854769e-007
  y^19/e^19x =   8.761417546430845e+006
  (y/e^x)^19 =   8.761417546430843e+006
  y^20/e^20x =                      NaN
  (y/e^x)^20 =   2.032230802424294e+007
```

Second, in order to prevent 'loss of significance', it is important to avoid a 'bad subtraction' (Section 1.2.2)—that is, a subtraction of a number from another number having almost equal value. Let us consider a simple problem of finding the roots of a second-order equation $ax^2 + bx + c = 0$ by using the quadratic formula

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{1.2.17}$$

Let $|4ac| \prec b^2$. Then, depending on the sign of b, a "bad subtraction" may be encountered when we try to find x_1 or x_2, which is the smaller one of the two roots. This implies that it is safe from the "loss of significance" to compute the root having the larger absolute value first and then obtain the other root by using the relation (between the roots and the coefficients) $x_1x_2 = c/a$.

For another instance, we consider the following two formulas, which are analytically the same, but numerically different:

$$f_1(x) = \frac{1 - \cos x}{x^2}, \qquad f_2(x) = \frac{\sin^2 x}{x^2(1 + \cos x)} \tag{1.2.18}$$

It is safe to use $f_1(x)$ for $x \approx \pi$ since the term $(1 + \cos x)$ in $f_2(x)$ is a 'bad subtraction', while it is safe to use $f_2(x)$ for $x \approx 0$ since the term $(1 - \cos x)$ in $f_1(x)$ is a 'bad subtraction'. Let's run the following MATLAB program "nm125_2.m" to confirm this. Below is the running result. This implies that we might use some formulas to avoid a 'bad subtraction'.

```
%nm125_2: round-off error test
f1 = inline('(1 - cos(x))/x/x','x');
f2 = inline('sin(x)*sin(x)/x/x/(1 + cos(x))','x');
for k = 0:1
  x = k*pi;  tmp = 1;
  for k1 = 1:8
    tmp = tmp*0.1; x1 = x + tmp;
    fprintf('At x = %10.8f, ', x1)
    fprintf('f1(x) = %18.12e; f2(x) = %18.12e', f1(x1),f2(x1));
  end
end
```

```
>> nm125_2
At x = 0.10000000, f1(x) = 4.995834721974e-001; f2(x) = 4.995834721974e-001
At x = 0.01000000, f1(x) = 4.999958333474e-001; f2(x) = 4.999958333472e-001
At x = 0.00100000, f1(x) = 4.999999583255e-001; f2(x) = 4.999999583333e-001
At x = 0.00010000, f1(x) = 4.999999969613e-001; f2(x) = 4.999999995833e-001
At x = 0.00001000, f1(x) = 5.000000413702e-001; f2(x) = 4.999999999958e-001
At x = 0.00000100, f1(x) = 5.000444502912e-001; f2(x) = 5.000000000000e-001
At x = 0.00000010, f1(x) = 4.996003610813e-001; f2(x) = 5.000000000000e-001
At x = 0.00000001, f1(x) = 0.000000000000e+000; f2(x) = 5.000000000000e-001
At x = 3.24159265, f1(x) = 1.898571371550e-001; f2(x) = 1.898571371550e-001
At x = 3.15159265, f1(x) = 2.013534055392e-001; f2(x) = 2.013534055391e-001
At x = 3.14259265, f1(x) = 2.025133720884e-001; f2(x) = 2.025133720914e-001
At x = 3.14169265, f1(x) = 2.026294667803e-001; f2(x) = 2.026294678432e-001
At x = 3.14160265, f1(x) = 2.026410772244e-001; f2(x) = 2.026410604538e-001
At x = 3.14159365, f1(x) = 2.026422382785e-001; f2(x) = 2.026242248740e-001
At x = 3.14159275, f1(x) = 2.026423543841e-001; f2(x) = 2.028044503269e-001
At x = 3.14159266, f1(x) = 2.026423659946e-001; f2(x) =                Inf
```

It may be helpful for avoiding a 'bad subtraction' to use the Taylor series expansion ([W-1]) rather than using the exponential function directly for the computation of e^x. For example, suppose we want to find

$$f_3(x) = \frac{e^x - 1}{x} \qquad \text{at } x = 0 \tag{1.2.19}$$

We can use the Taylor series expansion up to just the fourth-order of e^x about $x = 0$

$$\begin{aligned} g(x) = e^x &\approx g(0) + g'(0)x + \frac{g''(0)}{2!}x^2 + \frac{g^{(3)}(0)}{3!}x^3 + \frac{g^{(4)}(0)}{4!}x^4 \\ &= 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 \end{aligned}$$

to approximate the above function (1.2.19) as

$$f_3(x) = \frac{e^x - 1}{x} \approx 1 + \frac{1}{2!}x + \frac{1}{3!}x^2 + \frac{1}{4!}x^3 = f_4(x) \tag{1.2.20}$$

Noting that the true value of (1.2.9) is computed to be 1 by using the L'Hôpital's rule ([W-1]), we run the MATLAB program "`nm125_3.m`" to find which one of the two formulas $f_3(x)$ and $f_4(x)$ is better for finding the value of the expression (1.2.9) at $x = 0$. Would you compare them based on the running result shown below? How can the approximate formula $f_4(x)$ outrun the true one $f_3(x)$ for the numerical purpose, though not usual? It is because the zero factors in the numerator/denominator of $f_3(x)$ are canceled to set $f_4(x)$ free from the terror of a "bad subtraction."

```
%nm125_3: reduce the round-off error using Taylor series
f3 = inline('(exp(x)-1)/x','x');
f4 = inline('((x/4+1)*x/3) + x/2+1','x');
x = 0;  tmp = 1;
for k1 = 1:12
   tmp = tmp*0.1; x1 = x + tmp;
   fprintf('At x = %14.12f, ', x1)
   fprintf('f3(x) = %18.12e; f4(x) = %18.12e', f3(x1),f4(x1));
end
```

```
>> nm125_3
At x=0.100000000000, f3(x)=1.051709180756e+000; f4(x)=1.084166666667e+000
At x=0.010000000000, f3(x)=1.005016708417e+000; f4(x)=1.008341666667e+000
At x=0.001000000000, f3(x)=1.000500166708e+000; f4(x)=1.000833416667e+000
At x=0.000100000000, f3(x)=1.000050001667e+000; f4(x)=1.000083334167e+000
At x=0.000010000000, f3(x)=1.000005000007e+000; f4(x)=1.000008333342e+000
At x=0.000001000000, f3(x)=1.000000499962e+000; f4(x)=1.000000833333e+000
At x=0.000000100000, f3(x)=1.000000049434e+000; f4(x)=1.000000083333e+000
At x=0.000000010000, f3(x)=9.999999939225e-001; f4(x)=1.000000008333e+000
At x=0.000000001000, f3(x)=1.000000082740e+000; f4(x)=1.000000000833e+000
At x=0.000000000100, f3(x)=1.000000082740e+000; f4(x)=1.000000000083e+000
At x=0.000000000010, f3(x)=1.000000082740e+000; f4(x)=1.000000000008e+000
At x=0.000000000001, f3(x)=1.000088900582e+000; f4(x)=1.000000000001e+000
```

1.3 TOWARD GOOD PROGRAM

Among the various criteria about the quality of a general program, the most important one is how robust its performance is against the change of the problem properties and the initial values. A good program guides the program users who don't know much about the program and at least give them a warning message without runtime error for their minor mistake. There are many other features that need to be considered, such as user friendliness, compactness and elegance, readability, and so on. But, as far as the numerical methods are concerned, the accuracy of solution, execution speed (time efficiency), and memory utilization (space efficiency) are of utmost concern. Since some tips to achieve the accuracy or at least to avoid large errors (including overflow/underflow) are given in the previous section, we will look over the issues of execution speed and memory utilization.

1.3.1 Nested Computing for Computational Efficiency

The execution speed of a program for a numerical solution depends mostly on the number of function (subroutine) calls and arithmetic operations performed in the program. Therefore, we like the algorithm requiring fewer function calls and arithmetic operations. For instance, suppose we want to evaluate the value of a

polynomial

$$p_4(x) = a_1x^4 + a_2x^3 + a_3x^2 + a_4x + a_5 \tag{1.3.1}$$

It is better to use the nested structure (as below) than to use the above form as it is.

$$p_{4n}(x) = (((a_1x + a_2)x + a_3)x + a_4)x + a_5 \tag{1.3.2}$$

Note that the numbers of multiplications needed in Eqs. (1.3.2) and (1.3.1) are 4 and $(4 + 3 + 2 + 1 = 9)$, respectively. This point is illustrated by the program "`nm131_1.m`", where a polynomial $\sum_{i=0}^{N-1} a_i x^i$ of degree $N = 10^6$ for a certain value of x is computed by using the three methods—that is, Eq. (1.3.1), Eq. (1.3.2), and the MATLAB built-in function '`polyval()`'. Interested readers could run this program to see that Eq. (1.3.2)—that is, the nested multiplication—is the fastest, while '`polyval()`' is the slowest because of some overhead time for being called, though it is also fabricated in a nested structure.

```
%nm131_1: nested multiplication vs. plain multiple multiplication
N = 1000000+1; a = [1:N]; x = 1;
tic % initialize the timer
p = sum(a.*x.^[N-1:-1:0]); %plain multiplication
p, toc % measure the time passed from the time of executing 'tic'
tic, pn=a(1);
for i = 2:N %nested multiplication
   pn = pn*x + a(i);
end
pn, toc
tic, polyval(a,x), toc
```

Programming in a nested structure is not only recommended for time-efficient computation, but also may be critical to the solution. For instance, consider a problem of finding the value

$$S(K) = \sum_{k=0}^{K} \frac{\lambda^k}{k!} e^{-\lambda} \qquad \text{for } \lambda = 100 \quad \text{and} \quad K = 155 \tag{1.3.3}$$

```
%nm131_2_1: nested structure
lam = 100; K = 155;
p = exp(-lam);
S = 0;
for k = 1:K
  p=p*lam/k; S=S+p;
end
S
```

```
%nm131_2_2: not nested structure
lam = 100; K = 155;
S = 0;
for k = 1:K
  p = lam^k/factorial(k);
  S = S + p;
end
S*exp(-lam)
```

The above two programs are made for this computational purpose. Noting that this sum of Poisson probability distribution is close to 1 for such a large K, we

can run them to find that one works fine, while the other gives a quite wrong result. Could you tell which one is better?

1.3.2 Vector Operation Versus Loop Iteration

It is time-efficient to use vector operations rather than loop iterations to perform a repetitive job for an array of data. The following program "nm132_1.m" compares a vector operation versus a loop iteration in terms of the execution speed. Could you tell which one is faster?

```
%nm132_1: vector operation vs. loop iteration
N = 100000; th = [0:N-1]/50000*pi;
tic
ss=sin(th(1));
for i = 2:N, ss = ss + sin(th(i)); end  % loop iteration
toc, ss
tic
ss = sum(sin(th)); % vector operation
toc, ss
```

As a more practical example, let us consider a problem of finding the DtFT (discrete-time Fourier transform) ([W-3]) of a given sequence $x[n]$.

$$X(\Omega) = \sum_{n=0}^{N-1} x[n]e^{-j\Omega n} \qquad \text{for } \Omega = [-100:100]\pi/100 \tag{1.3.4}$$

The following program "nm132_2.m" compares a vector operation versus a loop iteration for computing the DtFT in terms of the execution speed. Could you tell which one is faster?

```
%nm132_2: nested structure
N = 1000; x = rand(1,N); % a random sequence x[n] for n = 0:N-1
W = [-100:100]*pi/100; % frequency range
tic
for k = 1:length(W)
  X1(k) = 0; %for for loop
  for n = 1:N, X1(k) = X1(k) + x(n)*exp(-j*W(k)*(n-1)); end
end
toc
tic
X2 = 0;
for n = 1:N  %for vector loop
  X2  = X2 +x(n)*exp(-j*W*(n-1));
end
toc
discrepancy = norm(X1-X2) %transpose for dimension compatibility
```

1.3.3 Iterative Routine Versus Nested Routine

In this section we compare an iterative routine and a nested routine performing the same job. Consider the following two programs `fctrl1(n)/fctrl2(n)`, whose common objectives is to get the factorial of a given nonnegative integer k.

$$k! = k(k-1)\cdots 2\cdot 1 \tag{1.3.5}$$

They differ in their structure. While `fctrl1()` uses a `for` loop structure, `fctrl2()` uses the nested (recursive) calling structure that a program uses itself as a subroutine to perform a sub-job. Compared with `fctrl1()`, `fctrl2()` is easier to program as well as to read, but is subject to runtime error that is caused by the excessive use of stack memory as the number of recursive calls increases with large `n`. Another disadvantage of `fctrl2()` is that it is time-inefficient for the number of function calls, which increases with the input argument (`n`). In this case, a professional programmer would consider the standpoint of users to determine the programming style. Some algorithms like the adaptive integration (Section 5.8), however, may fit the nested structure perfectly.

```
function m = fctrl1(n)
m = 1;
for k = 2:n, m = m*k; end
```

```
function m = fctrl2(n)
if n <= 1, m = 1;
 else  m = n*fctrl2(n-1);
end
```

1.3.4 To Avoid Runtime Error

A good program guides the program users who don't know much about the program and at least gives them a warning message without runtime error for their minor mistake. If you don't know what runtime error is, you can experience one by taking the following steps:

1. Make and save the above routine `fctrl1()` in an M-file named 'fctrl.m' in a directory listed in the MATLAB search path.
2. Type `fctrl(-1)` into the MATLAB Command window. Then you will see

```
>>fctrl(-1)
  ans = 1
```

This seems to imply that $(-1)! = 1$, which is not true. It is caused by the mistake of the user who tries to find $(-1)!$ without knowing that it is not defined. This kind of runtime error seems to be minor because it does not halt the process. But it needs special attention because it may not be easy to detect. If you are a good programmer, you will insert some error handling statements in the program `fctrl()` as below. Then, when someone happens to execute `fctrl(-1)` in the Command window or through an M-file, the execution stops and he will see the error message in the Command window as

```
??? Error using ==> fctrl
The factorial of negative number ??
```

```
function m = fctrl(n)
if n < 0, error('The factorial of negative number ??');
 else  m = 1; for k = 2:n, m = m*k; end
end
```

This shows the error message (given as the input argument of the `error()` routine) together with the name of the routine in which the accidental "error" happens, which is helpful for the user to avoid the error.

Most common runtime errors are caused by an "out of domain" index of array and the violation of matrix dimension compatibility, as illustrated in Section 1.1.7. For example, consider the `gauss(A,B)` routine in Section 2.2.2, whose job is to solve a system of linear equations `Ax = b` for `x`. To appreciate the role of the fifth line handling the dimension compatibility error in the routine, remove the line (by putting the comment mark `%` before the line in the M-file defining `gauss()`) and type the following statements in the Command window:

```
>>A = rand(3,3); B = rand(2,1); x = gauss(A,B)
    ?? Index exceeds matrix dimensions.
    Error in  ==> C:\MATLAB6p5\nma\gauss.m
    On line 10   ==> AB = [A(1:NA,1:NA) B(1:NA,1:NB)];
```

Then MATLAB gives you an error message together with the suspicious statement line and the routine name. But it is hard to figure out what causes the runtime error, and you may get nervous lest the routine should have some bug. Now, restore the fifth line in the routine and type the same statements in the Command window:

```
>>x = gauss(A,B)
    ?? Error using ==> gauss
    A and B must have compatible dimension
```

This error message (provided by the programmer of the routine) helps you to realize that the source of the runtime error is the incompatible matrices/vectors `A` and `B` given as the input arguments to the `gauss()` routine. Very like this, a good program has a scenario for possible user mistakes and fires the `error` routine for each abnormal condition to show the user the corresponding error message.

Many users often give more/fewer input arguments than supposed to be given to the MATLAB functions/routines and sometimes give wrong types/formats of data to them. To experience this type of error, let us try using the MATLAB function `sinc1(t,D)` (Section 1.3.5) to plot the graph of a sinc function

$$\sin c(t/D) = \frac{\sin(\pi t/D)}{\pi t/D} \text{ with } D = 0.5 \quad \text{and} \quad t = \left[-2, 2\right] \tag{1.3.6}$$

With this purpose, type the following statements in the Command window.

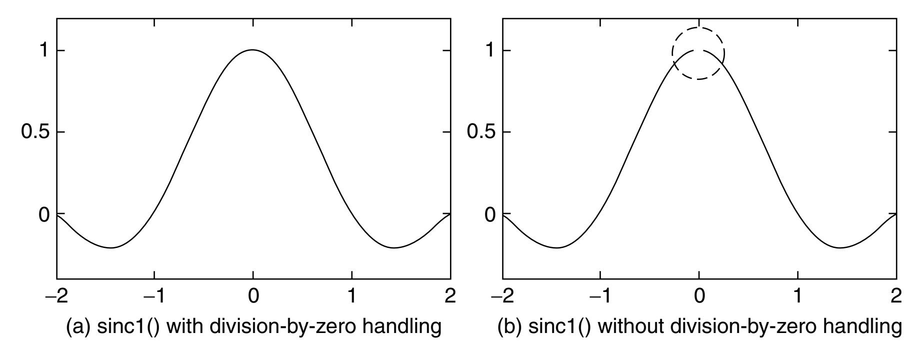


Figure 1.8 The graphs of a sinc function defined by `sinc1()`.

```
>>D = 0.5; b1 = -2; b2 = 2; t = b1+[0:200]/200*(b2 - b1);
>>plot(t,sinc1(t,D)), axis([b1 b2 -0.4 1.2])
>>hold on, plot(t,sinc1(t),'k:')
```

The two plotting commands coupled with `sinc1(t,D)` and `sinc1(t)` yield the two beautiful graphs, respectively, as depicted in Fig. 1.8a. It is important to note that `sinc1()` doesn't bother us and works fine without the second input argument `D`. We owe the second line in the function `sinc1()` for the nice error-handling service:

```
if nargin < 2, D = 1; end
```

This line takes care of the case where the number of input arguments (`nargin`) is less than 2, by assuming that the second input argument is `D = 1` by default. This programming technique is the key to making the MATLAB functions adaptive to different number/type of input arguments, which is very useful for breathing the user-convenience into the MATLAB functions. To appreciate its role, we remove the second line from the M-file defining `sinc1()` and then type the same statement in the Command window, trying to use `sinc1()` without the second input argument.

```
>>plot(t,sinc1(t),'k:')
  ??? Input argument 'D' is undefined.
  Error in ==> C:\MATLAB6p5\nma\sinc1.m
  On line 4 ==> x = sin(pi*t/D)./(pi*t/D);
```

This time we get a serious (red) error message with no graphic result. It is implied that the MATLAB function without the appropriate error-handling parts no longer allows the user's default or carelessness.

Now, consider the third line in `sinc1()`, which is another error-handling statement.

```
t(find(t==0))=eps;
```

or, equivalently

```
for i = 1:length(t), if t(i) == 0, t(i) = eps; end, end
```

This statement changes every zero element in the `t` vector into `eps` (2.2204e-016). What is the real purpose of this statement? It is actually to remove the possibility of division-by-zero in the next statement, which is a mathematical expression having `t` in the denominator.

```
x = sin(pi*t/D)./(pi*t/D);
```

To appreciate the role of the third line in `sinc1()`, we remove it from the M-file defining `sinc1()`, and type the following statement in the Command window.

```
>>plot(t,sinc1(t,D),'r')
   Warning: Divide by zero.
   (Type "warning off MATLAB:divideByZero" to suppress this warning.)
   In C:\MATLAB6p5\nma\sinc1.m at line 4)
```

This time we get just a warning (black) error message with a similar graphic result as depicted in Fig. 1.8b. Does it imply that the third line is dispensable? No, because the graph has a (weird) hole at $t = 0$, about which most engineers/mathematicians would feel uncomfortable. That's why authors strongly recommend you not to omit such an error-handling part as the third line as well as the second line in the MATLAB function `sinc1()`.

(cf) What is the value of `sinc1(t,D)` for `t = 0` in this case? Aren't you curious? If so, let's go for it.

```
>>sinc1(0,D), sin(pi*0/D)/(pi*0/D), 0/0
   ans = NaN (Not-a-Number: undetermined)
```

Last, consider of the fourth line in `sinc1()`, which is only one essential statement performing the main job.

```
x = sin(pi*t/D)./(pi*t/D);
```

What is the .(dot) before /(division operator) for? In reference to this, authors gave you a piece of advice that you had better put a .(dot) just before the arithmetic operators `*`(multiplication), `/`(division), and `^`(power) in the function definition so that the term-by-term (termwise) operation can be done any time (Section 1.1.6, (A5)). To appreciate the existence of the `.`(dot), we remove it from the M-file defining `sinc1()`, and type the following statements in the Command window.

```
>>clf, plot(t,sinc1(t,D)), sinc1(t,D), sin(pi*t/D)/(pi*t/D)
  ans = -0.0187
```

What do you see in the graphic window on the screen? Surprise, a (horizontal) straight line running parallel with the t-axis far from any sinc function graph! What is more surprising, the value of `sinc1(t,D)` or `sin(pi*t/D)/(pi*t/D)` shows up as a scalar. Authors hope that this accident will help you realize how important it is for right term-by-term operations to put .(dot) before the arithmetic operators `*`, `/` and `^` . By the way, aren't you curious about how MATLAB deals with a vector division without .(dot)? If so, let's try with the following statements:

```
>>A = [1:10]; B = 2*A; A/B, A*B'*(B*B')^-1, A*pinv(B)
    ans = 0.5
```

To understand this response of MATLAB, you can see Section 1.1.7 or Section 2.1.2.

In this section we looked at several sources of runtime error, hoping that it aroused the reader's attention to the danger of runtime error.

1.3.5 Parameter Sharing via Global Variables

When we discuss the runtime error that may be caused by user's default in passing some parameter as input argument to the corresponding function, you might feel that the parameter passing job is troublesome. Okay, it is understandable as a beginner in MATLAB. How about declaring the parameters as global so that they can be accessed/shared from anywhere in the MATLAB world as far as the declaration is valid? If you want to, you can declare any varable(s) by inserting the following statement in both the main program and all the functions using the variables.

```
global Gravity_Constant Dielectric_Constant
```

```
%plot_sinc
clear, clf
global D
D = 1; b1 = -2; b2 = 2;
t = b1 +[0:100]/100*(b2 - b1);
%passing the parameter(s) through arguments of the function
subplot(221), plot(t, sinc1(t,D))
axis([b1 b2 -0.4 1.2])
%passing the parameter(s) through global variables
subplot(222), plot(t, sinc2(t))
axis([b1 b2 -0.4 1.2])
```

```
function x = sinc1(t,D)
if nargin<2, D = 1; end
t(find(t == 0)) = eps;
x = sin(pi*t/D)./(pi*t/D);
```

```
function x = sinc2(t)
global D
t(find(t == 0)) = eps;
x = sin(pi*t/D)./(pi*t/D);
```

Then, how convenient it would be, since you don't have to bother about passing the parameters. But, as you get proficient in programming and handle many

functions/routines that are involved with various sets of parameters, you might find that the global variable is not always convenient, because of the following reasons.

- Once a variable is declared as global, its value can be changed in any of the MATLAB functions having declared it as global, without being noitced by other related functions. Therefore it is usual to declare only the constants as global and use long names (with all capital letters) as their names for easy identification.
- If some variables are declared as global and modified by several functions/routines, it is not easy to see the relationship and the interaction among the related functions in terms of the global variable. In other words, the program readability gets worse as the number of global variables and related functions increases.

For example, let us look over the above program "plot_sinc.m" and the function "`sinc2()`". They both have a declaration of `D` as global; consequently, `sinc2()` does not need the second input argument for getting the parameter `D`. If you run the program, you will see that the two plotting statements adopting `sinc1()` and `sinc2()` produce the same graphic result as depicted in Fig. 1.8a.

1.3.6 Parameter Passing Through Varargin

In this section we see two kinds of routines that get a function name (string) with its parameters as its input argument and play with the function.

First, let us look over the routine "`ez_plot1()`", which gets a function name (`ftn`) with its parameters (`p`) and the lower/upper bounds (`bounds = [b1 b2]`) as its first, third, and second input argument, respectively, and plots the graph of the given function over the interval set by the bounds. Since the given function may or may not have its parameter, the two cases are determined and processed by the number of input arguments (`nargin`) in the `if-else-end` block.

```
%plot_sinc1
clear, clf
D = 1; b1 = -2; b2 = 2;
t = b1+[0:100]/100*(b2 - b1);
bounds = [b1 b2];
subplot(223), ez_plot1('sinc1',bounds,D)
axis([b1 b2 -0.4 1.2])
subplot(224), ez_plot('sinc1',bounds,D)
axis([b1 b2 -0.4 1.2])
```

```
function ez_plot1(ftn,bounds,p)
if nargin < 2, bounds = [-1 1]; end
b1 = bounds(1); b2 = bounds(2);
t = b1+[0:100]/100*(b2 - b1);
if nargin <= 2, x = feval(ftn,t);
 else  x = feval(ftn,t,p);
end
plot(t,x)
```

```
function
ez_plot(ftn,bounds,varargin)
if nargin < 2, bounds = [-1 1]; end
b1 = bounds(1); b2 = bounds(2);
t = b1 + [0:100]/100*(b2 - b1);
x = feval(ftn,t,varargin{:});
plot(t,x)
```

Now, let us see the routine "`ez_plot()`", which does the same plotting job as "`ez_plot1()`". Note that it has a MATLAB keyword `varargin` (variable length argument list) as its last input argument and passes it into the MATLAB built-in function `feval()` as its last input argument. Since `varargin` can represent comma-separated multiple parameters including expression/strings, it paves the highway for passing the parameters in relays. As the number of parameters increases, it becomes much more convenient to use `varargin` for passing the parameters than to deal with the parameters one-by-one as in `ez_plot1()`. This technique will be widely used later in Chapter 4 (on nonlinear equations), Chapter 5 (on numerical integration), Chapter 6 (on ordinary differential equations), and Chapter 7 (on optimization).

(cf) Note that MATLAB has a built-in graphic function `ezplot()`, which is much more powerful and convenient to use than `ez_plot()`. You can type '`help ezplot`' to see its function and usage.

1.3.7 Adaptive Input Argument List

A MATLAB function/routine is said to be "adaptive" to users in terms of input arguments if it accepts different number/type of input arguments and makes a reasonable interpretation. For example, let us see the nonlinear equation solver routine '`newton()`' in Section 4.4. Its input argument list is

```
(f,df,x0,tol,kmax)
```

where `f, df, x0, tol` and `kmax` denote the filename (string) of function (to be solved), the filename (string) of its derivative function, the initial guess (for solution), the error tolerance and the maximum number of iterations, respectively. Suppose the user, not knowing the derivative, tries to use the routine with just four input arguments as follows.

```
>>newton(f,x0,tol,kmax)
```

At first, these four input arguments will be accepted as `f,df,x0,` and `tol`, respectively. But, when the second line of the program body is executed, the routine will notice something wrong from that `df` is not any filename but a number and then interprets the input arguments as `f,x0,tol,` and `kmax` to the idea of the user. This allows the user to use the routine in two ways, depending on whether he is going to supply the routine with the derivative function or not. This scheme is conceptually quite similar to function overloading of C++, but C++ requires us to have several functions having the same name, with different argument list.

PROBLEMS

1.1 Creating a Data File and Retrieving/Plotting Data Saved in a Data File

(a) Using the MATLAB editor, make a program "`nm1p01a`", which lets its user input data pairs of heights [ft] and weights [lb] of as many persons

as he wants until he presses <Enter> and save the whole data in the form of an $N \times 2$ matrix into an ASCII data file (***.dat) named by the user. If you have no idea how to compose such a program, you can permutate the statements in the box below to make your program. Store the program in the file named "nm1p01a.m" and run it to save the following data into the data file named "hw.dat":

5.5162
6.1185
5.7170
6.5195
6.2191

```
%nm1p01a: input data pairs and save them into an ASCII data file
clear
k = 0;
while 1
end
k = k + 1;
x(k,1) = h;
h = input('Enter height:')
x(k,2) = input('Enter weight:')
if isempty(h), break; end
cd('c:\matlab6p5\work') %change current working directory
filename = input('Enter filename(.dat):','s');
filename = [filename '.dat']; %string concatenation
save(filename,'x','/ascii')
```

(b) Make a MATLAB program "nm1p01b", which reads (loads) the data file "hw.dat" made in (a), plots the data as in Fig. 1.1a in the upper-left region of the screen divided into four regions like Fig. 1.3, and plots the data in the form of piecewise-linear (PWL) graph describing the relationship between the height and the weight in the upper-right region of the screen. Let each data pair be denoted by the symbol '+' on the graph. Also let the ranges of height and weight be [5, 7] and [160, 200], respectively. If you have no idea, you can permutate the statements in the below box. Additionally, run the program to check if it works fine.

```
%nm1p01b: to read the data file and plot the data
cd('c:\matlab6p5\work') %change current working directory
weight = hw(I,2);
load hw.dat
clf, subplot(221)
plot(hw)
subplot(222)
axis([5 7 160 200])
plot(height,weight,height,weight,'+')
[height,I] = sort(hw(:,1));
```

1.2 Text Printout of Alphanumeric Data

Make a routine `max_array(A)`, which uses the `max()` command to find one of the maximum elements of a matrix `A` given as its input argument and uses the `fprintf()` command to print it onto the screen together with its row/column indices in the following format.

```
'\n Max(A) is A(%2d,%2d) = %5.2f\n',row_index,col_index,maxA
```

Additionally, try it to have the maximum element of an arbitrary matrix (generated by the following two consecutive commands) printed in this format onto the screen.

```
>>rand('state',sum(100*clock)), rand(3)
```

1.3 Plotting the Mesh Graph of a Two-Dimensional Function

Consider the MATLAB program "`nm1p03a`", whose objective is to draw a cone.

(a) The statement on the sixth line seems to be dispensable. Run the program with and without this line and see what happens.

(b) If you want to plot the function fcone(x,y) defined in another M-file '`fcone.m`', how will you modify this program?

(c) If you replace the fifth line by '`Z = 1-abs(X)-abs(Y);`', what difference does it make?

```
%nm1p03a: to plot a cone
clear, clf
x = -1:0.02:1; y = -1:0.02:1;
[X,Y] = meshgrid(x,y);
Z = 1-sqrt(X.^2+Y.^2);
Z = max(Z,zeros(size(Z)));
mesh(X,Y,Z)
```

```
function z = fcone(x,y)
z = 1-sqrt(x.^2 + y.^2);
```

1.4 Plotting The Mesh Graph of Stratigraphic Structure

Consider the incomplete MATLAB program "`nm1p04`", whose objective is to draw a stratigraphic structure of the area around Pennsylvania State University from the several perspective point of view. The data about the depth of the rock layer at 5×5 sites are listed in Table P1.4. Supplement the incomplete parts of the program so that it serves the purpose and run the program to answer the following questions. If you complete it properly and run it, MATLAB will show you the four similar graphs at the four corners of the screen and be waiting for you to press any key.

(a) At what value of `k` does MATLAB show you the mesh/surface-type graphs that are the most similar to the first graphs? From this result, what do you guess are the default values of the azimuth or horizontal rotation angle and the vertical elevation angle (in degrees) of the perspective view point?

(b) As the first input argument `Az` of the command `view(Az,E1)` decreases, in which direction does the perspective viewpoint revolve round the z-axis, clockwise or counterclockwise (seen from the above)?

(c) As the second input argument `El` of the command `view(Az,E1)` increases, does the perspective viewpoint move up or down along the z-axis?

(d) What is the difference between the plotting commands `mesh()` and `meshc()`?

(e) What is the difference between the usages of the command `view()` with two input arguments `Az,El` and with a three-dimensional vector argument `[x,y,z]`?

Table P1.4 The Depth of the Rock Layer

	x Coordinate				
y Coordinate	0.1	1.2	2.5	3.6	4.8
0.5	410	390	380	420	450
1.4	395	375	410	435	455
2.2	365	405	430	455	470
3.5	370	400	420	445	435
4.6	385	395	410	395	410

```
%nm1p04: to plot a stratigraphic structure
clear, clf
x = [0.1 .. .. . ];
y = [0.5 .. .. . ];
Z = [410 390 .. .. .. .. ];
[X,Y] = meshgrid(x,y);
subplot(221), mesh(X,Y,500 - Z)
subplot(222), surf(X,Y,500 - Z)
subplot(223), meshc(X,Y,500 - Z)
subplot(224), meshz(X,Y,500 - Z)
pause
for k = 0:7
   Az = -12.5*k; El = 10*k; Azr = Az*pi/180; Elr = El*pi/180;
   subplot(221), view(Az,El)
   subplot(222),
   k, view([sin(Azr),-cos(Azr),tan(Elr)]), pause %pause(1)
end
```

1.5 Plotting a Function over an Interval Containing Its Singular Point Noting that the tangent function $f(x) = \tan(x)$ is singular at $x = \pi/2$, $3\pi/2$, let us plot its graph over $[0, 2\pi]$ as follows.

(a) Define the domain vector x consisting of sufficiently many intermediate point x_i's along the x-axis and the corresponding vector y consisting of the function values at x_i's and plot the vector y over the vector x. You may use the following statements.

```
>>x = [0:0.01:2*pi]; y = tan(x);
>>subplot(221), plot(x,y)
```

Which one is the most similar to what you have got, among the graphs depicted in Fig. P1.5? Is it far from your expectation?

(b) Expecting to get the better graph, we scale it up along the y-axis by using the following command.

```
>>axis([0 6.3 -10 10])
```

Which one is the most similar to what you have got, among the graphs depicted in Fig. P1.5? Is it closer to your expectation than what you got in (a)?

(c) Most probably, you must be nervous about the straight lines at the singular points $x = \pi/2$ and $x = 3\pi/2$. The more disturbed you become by the lines that must not be there, the better you are at the numerical stuffs. As an alternative to avoid such a singular happening, you can try dividing the interval into three sections excluding the two singular points as follows.

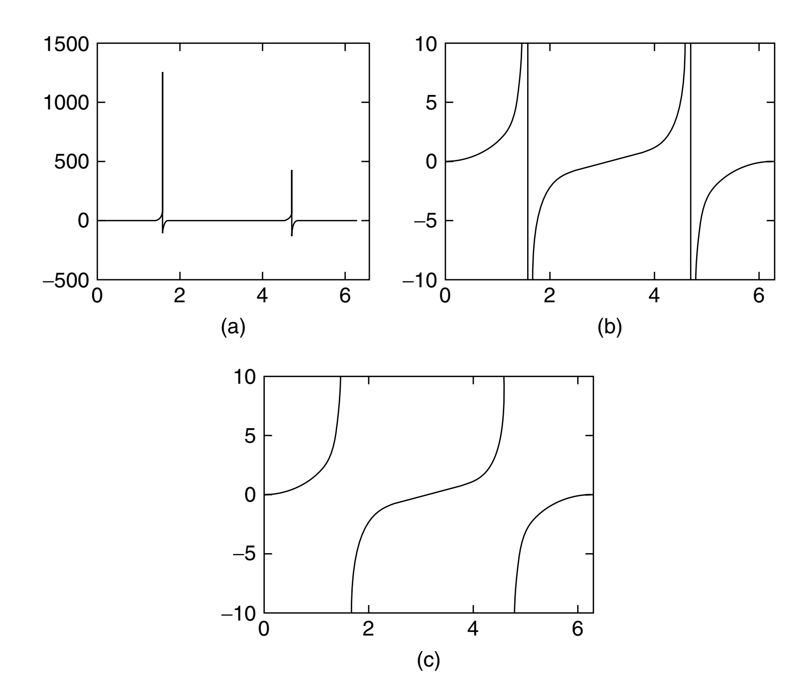


Figure P1.5 Plotting the graph of $f(x) = \tan x$.

```
>>x1 = [0:0.01:pi/2-0.01]; x2 = [pi/2+0.01:0.01:3*pi/2-0.01];
>>x3 = [3*pi/2+0.01:0.01:2*pi];
>>y1 = tan(x1); y2 = tan(x2); y3 = tan(x3);
>>subplot(222), plot(x1,y1,x2,y2,x3,y3), axis([0 6.3 -10 10])
```

(d) Try adjusting the number of intermediate points within the plotting interval as follows.

```
>>x1 = [0:200]*pi/100; y1 = tan(x1);
>>x2 = [0:400]*pi/200; y2 = tan(x2);
>>subplot(223), plot(x1,y1), axis([0 6.3 -10 10])
>>subplot(224), plot(x2,y2), axis([0 6.3 -10 10])
```

From the difference between the two graphs you got, you might have guessed that it would be helpful to increase the number of intermediate points. Do you still have the same idea even after you adjust the range of the y-axis to $[-50, +50]$ by using the following command?

```
>>axis([0 6.3 -50 50])
```

(e) How about trying the easy plotting command `ezplot()`? Does it answer your desire?

```
>>ezplot('tan(x)',0,2*pi)
```

1.6 Plotting the Graph of a Sinc Function

The sinc function is defined as

$$f(x) = \frac{\sin x}{x} \tag{P1.6.1}$$

whose value at $x = 0$ is

$$f(0) = \lim_{x\to 0} \frac{\sin x}{x} = \left.\frac{(\sin x)'}{x'}\right|_{x=0} = \left.\frac{\cos x}{1}\right|_{x=0} = 1 \tag{P1.6.2}$$

We are going to plot the graph of this function over $[-4\pi, +4\pi]$.

(a) Casually, you may try as follows.

```
>>x = [-100:100]*pi/25; y = sin(x)./x;
>>plot(x,y), axis([-15 15 -0.4 1.2])
```

In spite of the warning message about ‘division-by-zero’, you may somehow get a graph. But, is there anything odd about the graph?

(b) How about trying with a different domain vector?

```
>>x = [-4*pi:0.1:+4*pi]; y = sin(x)./x;
>>plot(x,y), axis([-15 15 -0.4 1.2])
```

Surprisingly, MATLAB gives us the function values without any complaint and presents a nice graph of the sinc function. What is the difference between (a) and (b)?

(cf) Actually, we would have no problem if we used the MATLAB built-in function `sinc()`.

1.7 Termwise (Element-by-Element) Operation in In-Line Functions

(a) Let the function $f_1(x)$ be defined without one or both of the dot(`.`) operators in Section 1.1.6. Could we still get the output vector consisting of the function values for the several values in the input vector? You can type the following statements into the MATLAB command window and see the results.

```
>>f1 = inline('1./(1+8*x^2)','x');  f1([0 1])
>>f1 = inline('1/(1+8*x.^2)','x');  f1([0 1])
```

(b) Let the function $f_1(x)$ be defined with both of the dot(`.`) operators as in Section 1.1.6. What would we get by typing the following statements into the MATLAB command window?

```
>>f1 = inline('1./(1+8*x.^2)','x');  f1([0 1]')
```

1.8 In-Line Function and M-file Function with the Integral Routine '`quad()`'

As will be seen in Section 5.8, one of the MATLAB built-in functions for computing the integral is '`quad()`', the usual usage of which is

$$\texttt{quad(f,a,b,tol,trace,p1,p2,..)} \quad \text{for} \quad \int_a^b f(x, p1, p2, \ldots)dx \tag{P1.8.1}$$

where

`f` is the name of the integrand function (M-file name should be categorized by `' '`)
`a,b` are the lower/upper bound of the integration interval
`tol` is the error tolerance (10^{-6} by default `[]`)
`trace` set to 1(on)/0(off) (0 by default `[]`) for subintervals
`p1,p2,..` are additional parameters to be passed directly to function f

Let's use this `quad()` routine with an in-line function and an M-file function to obtain

$$\int_{m-10}^{m+10} (x - x_0) f(x)dx \tag{P1.8.2a}$$

and

$$\int_{m-10}^{m+10} (x - x_0)^2 f(x)dx \tag{P1.8.2b}$$

where

$$x_0 = 1, \quad f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-m)^2/2\sigma^2} \quad \text{with } m = 1, \sigma = 2 \qquad \text{(P1.8.3)}$$

Below are an incomplete main program 'nm1p08' and an M-file function defining the integrand of (P1.8.2a). Make another M-file defining the integrand of (P1.8.2b) and complete the main program to compute the two integrals (P1.8.2a) and (P1.8.2b) by using the in-line/M-file functions.

```
function xfx = xGaussian_pdf(x,m,sigma,x0)
xfx = (x - x0).*exp(-(x - m).^2/2/sigma^2)/sqrt(2*pi)/sigma;
```

```
%nm1p08: to try using quad() with in-line/M-file functions
clear
m = 1; sigma = 2;
int_xGausspdf = quad('xGaussian_pdf',m - 10,m + 10,[],0,m,sigma,1)
Gpdf = 'exp(-(x-m).^2/2/sigma^2)/sqrt(2*pi)/sigma';
xGpdf = inline(['(x - x0).*' Gpdf],'x','m','sigma','x0');
int_xGpdf = quad(xGpdf,m - 10,m+10,[],0,m,sigma,1)
```

1.9 μ-Law Function Defined in an M-File

The so-called μ-law function and μ^{-1}-law function used for non-uniform quantization is defined as

$$y = g_\mu(x) = |y|_{\max} \frac{\ln(1 + \mu|x|/|x|_{\max})}{\ln(1 + \mu)} \text{sign}(x) \qquad \text{(P1.9a)}$$

$$x = g_\mu^{-1}(y) = |x|_{\max} \frac{(1 + \mu)^{|y|/|y|_{\max}} - 1}{\mu} \text{sign}(y) \qquad \text{(P1.9b)}$$

Below are the μ-law function `mulaw()` defined in an M-file and a main program `nm1p09`, which performs the following jobs:

- Finds the values `y` of the μ-law function for `x = [-1:0.01:1]`, plots the graph of `y` versus `x`.
- Finds the values x0 of the μ^{-1}-law function for `y`.
- Computes the discrepancy between `x` and `x0`.

Complete the μ^{-1}-law function `mulaw_inv()` and store it together with `mulaw()` and `nm1p09` in the M-files named "mulaw_inv.m", "mulaw.m", and "nm1p09.m", respectively. Then run the main program `nm1p09` to plot the graphs of the μ-law function with $\mu = 10$, 50 and 255 and find the discrepancy between `x` and `x0`.

```
function [y,xmax] = mulaw(x,mu,ymax)
xmax = max(abs(x));
y = ymax*log(1+mu*abs(x/xmax))./log(1+mu).*sign(x);  % Eq.(P1.9a)
```

```
function x = mulaw_inv(y,mu,xmax)
```

```
%nm1p09: to plot the mulaw curve
clear, clf
x = [-1:.005:1];
mu = [10 50 255];
for i = 1:3
  [y,xmax] = mulaw(x,mu(i),1);
  plot(x,y,'b-', x,x0,'r-'), hold on
  x0 = mulaw_inv(y,mu(i),xmax);
  discrepancy = norm(x-x0)
end
```

1.10 Analog-to-Digital Converter (ADC)

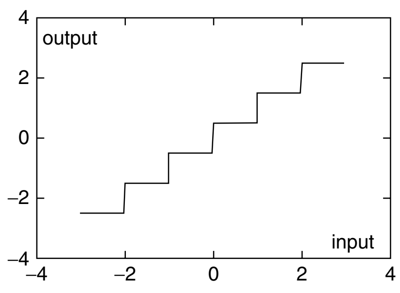
Below are two ADC routines `adc1(a,b,c)` and `adc2(a,b,c)`, which assign the corresponding digital value `c(i)` to each one of the analog data belonging to the quantization interval [`b(i)`, `b(i+1)`]. Let the boundary vector and the centroid vector be, respectively,

```
b = [-3 -2 -1 0 1 2 3];     c = [-2.5 -1.5 -0.5 0.5 1.5 2.5];
```

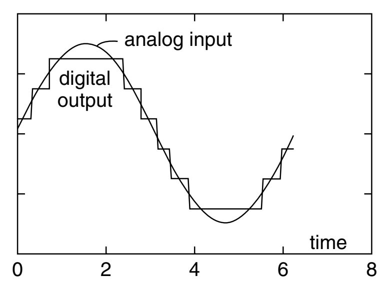
(a) Make a program that uses two ADC routines to find the output `d` for the analog input data `a = [-300:300]/100` and plots `d` versus `a` to see the input-output relationship of the ADC, which is supposed to be like Fig. P1.10a.

```
function d = adc1(a,b,c)
%Analog-to-Digital Converter
%Input   a = analog signal, b(1:N + 1) = boundary vector
         c(1:N)=centroid vector
%Output: d = digital samples
N = length(c);
for n = 1:length(a)
  I = find(a(n) < b(2:N));
  if ~isempty(I), d(n) = c(I(1));
   else           d(n) = c(N);
  end
end
```

```
function d=adc2(a,b,c)
N = length(c);
d(find(a < b(2))) = c(1);
for i = 2:N-1
  index = find(b(i) <= a & a <= b(i+1));  d(index) = c(i);
end
d(find(b(N) <= a)) = c(N);
```



(a) The input-output relationship of an ADC



(b) The output of an ADC to a sinusoidal input

Figure P1.10 The characteristic of an ADC (analog-to-digital converter).

(b) Make a program that uses two ADC routines to find the output `d` for the analog input data `a = 3*sin(t)` with `t = [0:200]/100*pi` and plots `a` and `d` versus `t` to see how the analog input is converted into the digital output by the ADC. The graphic result is supposed to be like Fig. P1.10b.

1.11 Playing with Polynomials

(a) Polynomial Evaluation: `polyval()`
Write a MATLAB statement to compute

$$p(x) = x^8 - 1 \qquad \text{for } x = 1 \tag{P1.11.1}$$

(b) Polynomial Addition/Subtraction by Using Compatible Vector Addition/Subtraction
Write a MATLAB statement to add the following two polynomials:

$$p_1(x) = x^4 + 1, \qquad p_2(x) = x^3 - 2x^2 + 1 \tag{P1.11.2}$$

(c) Polynomial Multiplication: `conv()`
Write a MATLAB statement to get the following product of polynomials:

$$p(x) = (x^4 + 1)(x^2 + 1)(x + 1)(x - 1) \tag{P1.11.3}$$

(d) Polynomial Division: `deconv()`
Write a MATLAB statement to get the quotient and the remainder of the following polynomial division:

$$p(x) = x^8/(x^2 - 1) \tag{P1.11.4}$$

(e) Routine for Differentiation/Integration of a Polynomial
What you see in the below box is the routine "`poly_der(p)`", which gets a polynomial coefficient vector `p` (in the descending order) and outputs the coefficient vector `pd` of its derivative polynomial. Likewise, you can make a routine "`poly_int(p)`", which outputs the coefficient

vector of the integral polynomial for a given polynomial coefficient vector.

(cf) MATLAB has the built-in routines `polyder()/polyint()` for finding the derivative/integral of a polynomial.

```
function  pd = poly_der(p)
%p: the vector of polynomial coefficients in descending order
N = length(p);
if N <= 1,  pd = 0; % constant
else
  for i = 1: N - 1, pd(i)  = p(i)*(N - i); end
end
```

(f) Roots of A Polynomial Equation: `roots()`
Write a MATLAB statement to get the roots of the following polynomial equation

$$p(x) = x^8 - 1 = 0 \tag{P1.11.5}$$

You can check if the result is right, by using the MATLAB command `poly()`, which generates a polynomial having a given set of roots.

(g) Partial Fraction Expansion of a Ratio of Two Polynomials: `residue()/residuez()`

(i) The MATLAB routine `[r,p,k] = residue(B,A)` finds the partial fraction expansion for a ratio of given polynomials $B(s)/A(s)$ as

$$\frac{B(s)}{A(s)} = \frac{b_1 s^{M-1} + b_2 s^{M-2} + \cdots + b_M}{a_1 s^{N-1} + a_2 s^{N-2} + \cdots + a_N} = k(s) + \sum_i \frac{r(i)}{s - p(i)} \tag{P1.11.6a}$$

which is good for taking the inverse Laplace transform. Use this routine to find the partial fraction expansion for

$$X(s) = \frac{4s + 2}{s^3 + 6s^2 + 11s + 6} = \frac{\quad}{s + \quad} + \frac{\quad}{s + \quad} + \frac{\quad}{s + \quad} \tag{P1.11.7a}$$

(ii) The MATLAB routine `[r,p,k] = residuez(B,A)` finds the partial fraction expansion for a ratio of given polynomials $B(z)/A(z)$ as

$$\frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \cdots + b_M z^{-(M-1)}}{a_1 + a_2 z^{-1} + \cdots + a_N z^{-(N-1)}} = k(z^{-1}) + \sum_i \frac{r(i) z}{z - p(i)} \tag{P1.11.6b}$$

which is good for taking the inverse z-transform. Use this routine to find the partial fraction expansion for

$$X(z) = \frac{4 + 2z^{-1}}{1 + 6z^{-1} + 11z^{-2} + 6z^{-3}} = \frac{z}{z + \quad} + \frac{z}{z + \quad} + \frac{z}{z + \quad} \tag{P1.11.7b}$$

(h) Piecewise Polynomial: `mkpp()/ppval()`
Suppose we have an $M \times N$ matrix `P`, the rows of which denote M (piecewise) polynomials of degree $(N-1)$ for different (non-overlapping) intervals with `(M + 1)` boundary points `bb = [b(1) .. b(M + 1)]`, where the polynomial coefficients in each row are supposed to be generated with the interval starting from $x = 0$. Then we can use the MATLAB command `pp = mkpp(bb,P)` to construct a structure of piecewise polynomials, which can be evaluated by using `ppval(pp)`.

Figure P1.11(h) shows a set of piecewise polynomials $\{p_1(x+3), p_2(x+1), p_3(x-2)\}$ for the intervals [−3, −1],[−1, 2] and [2, 4], respectively, where

$$p_1(x) = x^2, \quad p_2(x) = -(x-1)^2, \quad \text{and} \quad p_3(x) = x^2 - 2 \qquad \text{(P1.11.8)}$$

Make a MATLAB program which uses `mkpp()/ppval()` to plot this graph.

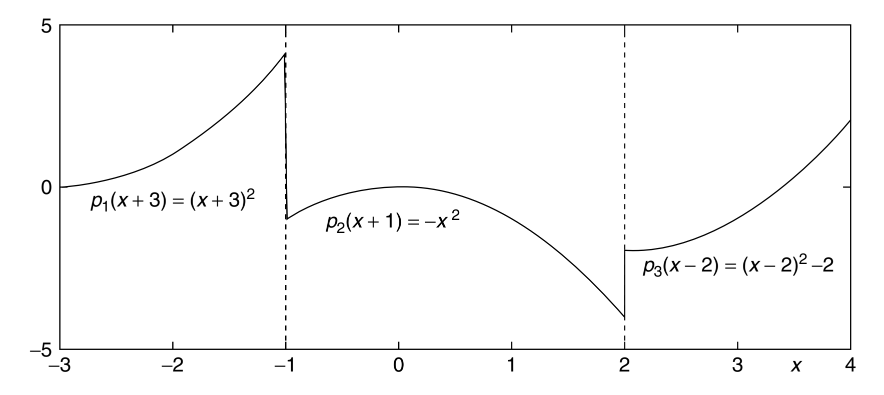


Figure P1.11(h) The graph of piecewise polynomial functions.

(cf) You can type '`help mkpp`' to see a couple of examples showing the usage of `mkpp`.

1.12 Routine for Matrix Multiplication

Assuming that MATLAB cannot perform direct multiplication on vectors/matrices, supplement the following incomplete routine "`multiply_matrix(A,B)`" so that it can multiply two matrices given as its input arguments only if their dimensions are compatible, but displays an error message if their dimensions are not compatible. Try it to get the product of two arbitrary 3×3 matrices generated by the command `rand(3)` and compare the result with that obtained by using the direct multiplicative operator `*`. Note that

the matrix multiplication can be described as

$$C(m,n) = \sum_{k=1}^{K} A(m,k)B(k,n) \tag{P1.12.1}$$

```
function C = multiply_matrix(A,B)
[M,K] = size(A);  [K1,N] = size(B);
if K1 ~= K
  error('The # of columns of A is not equal to the # of rows of B')
else
  for m = 1:
     for n = 1:
        C(m,n) = A(m,1)*B(1,n);
        for k = 2:
           C(m,n) = C(m,n) + A(m,k)*B(k,n);
        end
     end
  end
end
```

1.13 Function for Finding Vector Norm

Assuming that MATLAB does not have the `norm()` command finding us the norm of a given vector/matrix, make a routine `norm_vector(v,p)`, which computes the norm of a given vector as

$$\|\mathbf{v}\|_p = \sqrt[p]{\sum_{n=1}^{N} |v_n|^p} \tag{P1.13.1}$$

for any positive integer `p`, finds the maximum absolute value of the elements for `p = inf` and computes the norm as if `p = 2`, even if the second input argument `p` is not given. If you have no idea, permutate the statements in the below box and save it in the file named "`norm_vector.m`". Additionally, try it to get the norm with `p = 1,2,`∞ (`inf`) and of an arbitrary vector generated by the command `rand(2,1)`. Compare the result with that obtained by using the `norm()` command.

```
function nv = norm_vector(v,p)
if nargin < 2, p = 2; end
nv = sum(abs(v).^p)^(1/p);
nv = max(abs(v));
if p > 0 & p ~= inf
 elseif p == inf
end
```

1.14 Backslash(`\`) Operator

Let's play with the backslash(\) operator.

(a) Use the backslash(\) command, the minimum-norm solution (2.1.7) and the `pinv()` command to solve the following equations, find the residual error $\|A_i\mathbf{x} - \mathbf{b}_i\|$'s and the rank of the coefficient matrix A_i, and fill in Table P1.14 with the results.

$$\textbf{(i)}\ A_1\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 15 \end{bmatrix} = \mathbf{b}_1 \tag{P1.14.1}$$

$$\textbf{(ii)}\ A_2\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 8 \end{bmatrix} = \mathbf{b}_2 \tag{P1.14.2}$$

$$\textbf{(iii)}\ A_3\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 12 \end{bmatrix} = \mathbf{b}_3 \tag{P1.14.3}$$

Table P1.14 Results of Operations with `backslash` (\) Operator and `pinv( )` Command

	`backslash(\)`		Minimum-Norm or LS Solution		`pinv()`		Remark on $rank(A_i)$
	$\mathbf{x}$	$\|A_i\mathbf{x} - \mathbf{b}_i\|$	$\mathbf{x}$	$\|A_i\mathbf{x} - \mathbf{b}_i\|$	$\mathbf{x}$	$\|A_i\mathbf{x} - \mathbf{b}_i\|$	redundant/ inconsistent
$A_1\mathbf{x} = \mathbf{b}_1$	1.5000 0 1.5000	4.4409e-15 (1.9860e-15)					
$A_2\mathbf{x} = \mathbf{b}_2$					0.3143 0.6286 0.9429	1.7889	
$A_3\mathbf{x} = \mathbf{b}_3$							
$A_4\mathbf{x} = \mathbf{b}_4$			2.5000 0.0000	1.2247			
$A_5\mathbf{x} = \mathbf{b}_5$							
$A_6\mathbf{x} = \mathbf{b}_6$							

(cf) When the mismatching error $\|A_i\mathbf{x} - \mathbf{b}_i\|$'s obtained from MATLAB 5.x/6.x version are slightly different, the former one is in the parentheses ().

(b) Use the backslash (\) command, the LS (least-squares) solution (2.1.10) and the `pinv()` command to solve the following equations and find the residual error $\|A_i\mathbf{x} - \mathbf{b}_i\|$'s and the rank of the coefficient matrix A_i, and fill in Table P1.14 with the results.

$$\textbf{(i)}\ A_4\mathbf{x} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 7 \end{bmatrix} = \mathbf{b}_4 \tag{P1.14.4}$$

$$\textbf{(ii)}\ A_5\mathbf{x} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 8 \end{bmatrix} = \mathbf{b}_5 \tag{P1.14.5}$$

$$\textbf{(iii)}\ A_6\mathbf{x} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix} = \mathbf{b}_6 \tag{P1.14.6}$$

(cf) If some or all of the rows of the coefficient matrix A in a set of linear equations can be expressed as a linear combination of other row(s), the corresponding equations are dependent, which can be revealed by the rank deficiency, that is, $\text{rank}(A) < \min(M, N)$ where M and N are the row dimension and the column dimension, respectively. If some equations are dependent, they may have either inconsistency (no exact solution) or redundancy (infinitely many solutions), which can be distinguished by checking if augmenting the RHS vector **b** to the coefficient matrix A increases the rank or not—that is, $\text{rank}([A\ \mathbf{b}]) > \text{rank}(A)$ or not [M-2].

(c) Based on the results obtained in (a) and (b) and listed in Table P1.14, answer the following questions.

(i) Based on the results obtained in (a)(i), which one yielded the non-minimum-norm solution among the three methods, that is, the backslash(\) operator, the minimum-norm solution (2.1.7) and the `pinv()` command? Note that the minimum-norm solution means the solution whose norm ($\|\mathbf{x}\|$) is the minimum over the many solutions.

(ii) Based on the results obtained in (a), which one is most reliable as a means of finding the minimum-norm solution among the three methods?

(iii) Based on the results obtained in (b), choose two reliable methods as a means of finding the LS (least-squares) solution among the three methods, that is, the backslash (\) operator, the LS solution (2.1.10) and the `pinv()` command. Note that the LS solution

means the solution for which the residual error ($\|A\mathbf{x} - \mathbf{b}\|$) is the minimum over the many solutions.

1.15 Operations on Vectors

(a) Find the mathematical expression for the computation to be done by the following MATLAB statements.

```
>>n = 0:100; S = sum(2.^-n)
```

(b) Write a MATLAB statement that performs the following computation.

$$\left(\sum_{n=0}^{10000} \frac{1}{(2n+1)^2}\right) - \frac{\pi^2}{8}$$

(c) Write a MATLAB statement which uses the commands `prod()` and `sum()` to compute the product of the sums of each row of a 3×3 random matrix.

(d) How does the following MATLAB routine "`repetition(x,M,m)`" convert a given row vector sequence `x` to make a new sequence `y` ?

```
function y = repetition(x,M,m)
if m == 1
 MNx = ones(M,1)*x; y = MNx(:)';
else
 Nx = length(x); N = ceil(Nx/m);
 x = [x zeros(1,N*m - Nx)];
 MNx = ones(M,1)*x;
 y = [];
 for n = 1:N
   tmp = MNx(:,(n - 1)*m + [1:m]).';
   y = [y tmp(:).'];
 end
end
```

(e) Make a MATLAB routine "`zero_insertion(x,M,m)`", which inserts `m` zeros just after every `M`th element of a given row vector sequence `x` to make a new sequence. Write a MATLAB statement to apply the routine for inserting two zeros just after every third element of `x` $=$ $[1 \quad 3 \quad 7 \quad 2 \quad 4 \quad 9]$ to get

$$\mathtt{y} = [1 \quad 3 \quad 7 \quad 0 \quad 0 \quad 2 \quad 4 \quad 9 \quad 0 \quad 0]$$

(f) How does the following MATLAB routine "`zeroing(x,M,m)`" convert a given row vector sequence `x` to make a new sequence `y`?

```
function y = zeroing(x,M,m)
%zero out every (kM - m)th element
if nargin < 3, m = 0; end
if M<=0, M = 1; end
m = mod(m,M);
Nx = length(x); N = floor(Nx/M);
y = x; y(M*[1:N] - m) = 0;
```

(g) Make a MATLAB routine "`sampling(x,M,m)`", which samples every (`kM - m`)th element of a given row vector sequence `x` to make a new sequence. Write a MATLAB statement to apply the routine for sampling every $(3k-2)$th element of $x = [1 \quad 3 \quad 7 \quad 2 \quad 4 \quad 9]$ to get

$$y = [1\ 2]$$

(h) Make a MATLAB routine '`rotation_r(x,M)`", which rotates a given row vector sequence `x` right by `M` samples, say, making `rotate_r([1 2 3 4 5],3) = [3 4 5 1 2]`.

1.16 Distribution of a Random Variable: Histogram

Make a routine `randu(N,a,b)`, which uses the MATLAB function `rand()` to generate an `N`-dimensional random vector having the uniform distribution over [`a`, `b`] and depicts the graph for the distribution of the elements of the generated vector in the form of histogram divided into 20 sections as Fig.1.7. Then, see what you get by typing the following statement into the MATLAB command window.

```
>>randu(1000,-2,2)
```

What is the height of the histogram on the average?

1.17 Number Representation

In Section 1.2.1, we looked over how a number is represented in 64 bits. For example, the IEEE 64-bit floating-point number system represents the number $3(2^1 \le 3 < 2^2)$ belonging to the range $R_1 = [2^1, 2^2)$ with $E = 1$ as

0	100 0000 0000	1000 0000 0000 . . . . 0000 0000 0000 0000 0000

4 0 0 8 0 0 . . . . 0 0 0 0 0

where the exponent and the mantissa are

$$\text{Exp} = \text{E} + 1023 = 1 + 1023 = 1024 = 2^{10} = 100\ 0000\ 0000$$

$$\text{M} = (3 \times 2^{-\text{E}} - 1) \times 2^{52} = 2^{51}$$

$$= 1000\ 0000\ 0000 \ldots .\ 0000\ 0000\ 0000\ 0000\ 0000$$

This can be confirmed by typing the following statement into MATLAB command window.

```
>>fprintf('3 = %bx\n',3)    or    >>format hex, 3, format short
```

which will print out onto the screen

```
0000000000000840              4008000000000000
```

Noting that more significant byte (8[bits] = 2[hexadecimal digits]) of a number is stored in the memory of higher address number in the INTEL system, we can reverse the order of the bytes in this number to see the number having the most/least significant byte on the left/right side as we can see in the daily life.

```
00 00 00 00 00 00 08 40 → 40 08 00 00 00 00 00 00
```

This is exactly the hexadecimal representation of the number 3 as we expected. You can find the IEEE 64-bit floating-point number representation of the number 14 and use the command `fprintf()` or format hex to check if the result is right.

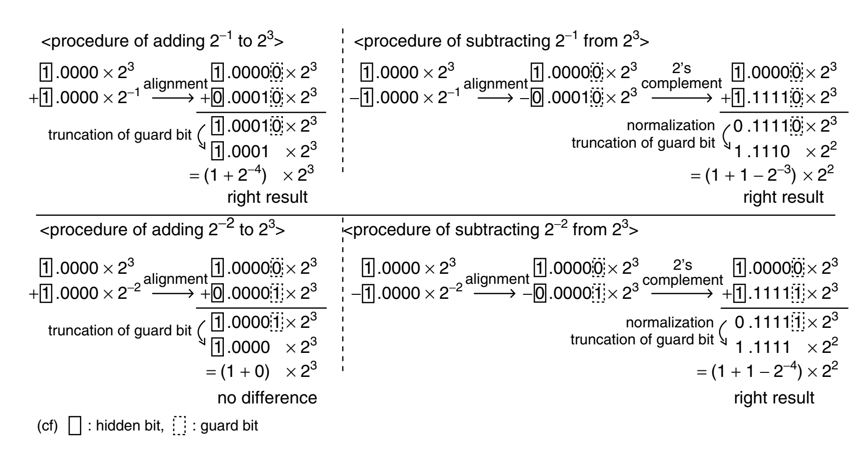


Figure P1.18 Procedure of addition/subtraction with four mantissa bits.

1.18 Resolution of Number Representation and Quantization Error

In Section 1.2.1, we have seen that adding 2^{-22} to 2^{30} makes some difference, while adding 2^{-23} to 2^{30} makes no difference due to the bit shift by over 52 bits for alignment before addition. How about subtracting 2^{-23} from 2^{30}? In contrast with the addition of 2^{-23} to 2^{30}, it makes a difference as you can see by typing the following statement into the MATLAB

command window.

```
>>x = 2^30; x + 2^ - 23 == x, x - 2^ - 23 == x
```

which will give you the logical answer 1 (true) and 0 (false). Justify this result based on the difference of resolution of two ranges $[2^{30}, 2^{31})$ and $[2^{29}, 2^{30})$ to which the true values of computational results $(2^{30} + 2^{-23})$ and $(2^{30} - 2^{-23})$ belong, respectively. Note from Eq. (1.2.5) that the resolutions—that is, the maximum quantization errors—are $\Delta_E = 2^{E-52} = 2^{-52+30} = 2^{-22}$ and $2^{-52+29} = 2^{-23}$, respectively. For details, refer to Fig. P1.18, which illustrates the procedure of addition/subtraction with four mantissa bits, one hidden bit, and one guard bit.

1.19 Resolution of Number Representation and Quantization Error

(a) What is the result of typing the following statements into the MATLAB command window?

```
>>7/100*100 - 7
```

How do you compare the absolute value of this answer with the resolution Δ of the range to which 7 belongs?

(b) Find how many numbers are susceptible to this kind of quantization error caused by division/multiplication by 100, among the numbers from 1 to 31.

(c) What will be the result of running the following program? Why?

```
%nm1p19: Quantization Error
x = 2-2^-50;
for n = 1:2^3
   x = x+2^-52; fprintf('%20.18E\n',x)
end
```

1.20 Avoiding Large Errors/Overflow/Underflow

(a) For $x = 9.8^{201}$ and $y = 10.2^{199}$, evaluate the following two expressions that are mathematically equivalent and tell which is better in terms of the power of resisting the overflow.

(i) $z = \sqrt{x^2 + y^2}$ (P1.20.1a)

(ii) $z = y\sqrt{(x/y)^2 + 1}$ (P1.20.1b)

Also for $x = 9.8^{-201}$ and $y = 10.2^{-199}$, evaluate the above two expressions and tell which is better in terms of the power of resisting the underflow.

(b) With $a = c = 1$ and for 100 values of b over the interval $[10^{7.4}, 10^{8.5}]$ generated by the MATLAB command ‘`logspace(7.4,8.5,100)`’,

evaluate the following two formulas (for the roots of a quadratic equation) that are mathematically equivalent and plot the values of the second root of each pair. Noting that the true values are not available and so the shape of solution graph is only one practical basis on which we can assess the quality of numerical solutions, tell which is better in terms of resisting the loss of significance.

(i) $$\left[x_1, x_2 = \frac{1}{2a}(-b \mp \text{sign}(b)\sqrt{b^2 - 4ac})\right] \tag{P1.20.2a}$$

(ii) $$\left[x_1 = \frac{1}{2a}(-b - \text{sign}(b)\sqrt{b^2 - 4ac}),\ x_2 = \frac{c/a}{x_1}\right] \tag{P1.20.2b}$$

(c) For 100 values of x over the interval $[10^{14}, 10^{16}]$, evaluate the following two expressions that are mathematically equivalent, plot them, and based on the graphs, tell which is better in terms of resisting the loss of significance.

(i) $$y = \sqrt{2x^2 + 1} - 1 \tag{P1.20.3a}$$

(ii) $$y = \frac{2x^2}{\sqrt{2x^2 + 1} + 1} \tag{P1.20.3b}$$

(d) For 100 values of x over the interval $[10^{-9}, 10^{-7.4}]$, evaluate the following two expressions that are mathematically equivalent, plot them, and based on the graphs, tell which is better in terms of resisting the loss of significance.

(i) $$y = \sqrt{x + 4} - \sqrt{x + 3} \tag{P1.20.4a}$$

(ii) $$y = \frac{1}{\sqrt{x + 4} + \sqrt{x + 3}} \tag{P1.20.4b}$$

(e) On purpose to find the value of $(300^{125}/125!)e^{-300}$, type the following statement into the MATLAB command window.

```
>>300^125/prod([1:125])*exp(-300)
```

What is the result? Is it of any help to change the order of multiplication/division? As an alternative, make a routine which evaluates the expression

$$p(k) = \frac{\lambda^k}{k!}e^{-\lambda} \qquad \text{for } \lambda = 300 \text{ and an integer } k \tag{P1.20.5}$$

in a recursive way, say, like $p(k + 1) = p(k) * \lambda/k$ and then, use the routine to find the value of $(300^{125}/125!)e^{-300}$.

(f) Make a routine which computes the sum

$$S(K) = \sum_{k=0}^{K} \frac{\lambda^k}{k!} e^{-\lambda} \qquad \text{for } \lambda = 100 \text{ and an integer } K \qquad \text{(P1.20.6)}$$

and then, use the routine to find the value of $S(155)$.

1.21 Recursive Routines for Efficient Computation

(a) The Hermite Polynomial [K-1]
Consider the Hermite polynomial defined as

$$H_0(x) = 1, \qquad H_N(x) = (-1)^N e^{x^2} \frac{d^N}{dx^N} e^{-x^2} \qquad \text{(P1.21.1)}$$

(i) Show that the derivative of this polynomial function can be written as

$$\begin{aligned} H'_N(x) &= (-1)^N 2x e^{x^2} \frac{d^N}{dx^N} e^{-x^2} + (-1)^N e^{x^2} \frac{d^{N+1}}{dx^{N+1}} e^{-x^2} \\ &= 2x H_N(x) - H_{N+1}(x) \end{aligned} \qquad \text{(P1.21.2)}$$

and so the $(N+1)$th-degree Hermite polynomial can be obtained recursively from the Nth-degree Hermite polynomial as

$$H_{N+1}(x) = 2x H_N(x) - H'_N(x) \qquad \text{(P1.21.3)}$$

(ii) Make a MATLAB routine "`Hermitp(N)`" which uses Eq. (P1.21.3) to generate the Nth-degree Hermite polynomial $H_N(x)$.

(b) The Bessel Function of the First Kind [K-1]
Consider the Bessel function of the first kind of order k defined as

$$J_k(\beta) = \frac{1}{\pi} \int_0^{\pi} \cos(k\delta - \beta \sin \delta) d\delta \qquad \text{(P1.21.4a)}$$

$$= \left(\frac{\beta}{2}\right)^k \sum_{m=0}^{\infty} \frac{(-1)^m \beta^{2m}}{4^m m!(m+k)!} \equiv (-1)^k J_{-k}(\beta) \qquad \text{(P1.21.4b)}$$

(i) Define the integrand of (P1.21.4a) in the name of '`Bessel_integrand(x,beta,k)`' and store it in an M-file named "`Bessel_integrand.m`".

(ii) Complete the following routine "`Jkb(K,beta)`", which uses (P1.21.4b) in a recursive way to compute $J_k(\beta)$ of order $k =$ `1:K` for given `K` and β (`beta`).

(iii) Run the following program `nm1p21b` which uses Eqs. (P1.21.4a) and (P1.21.4b) to get $J_{15}(\beta)$ for $\beta = 0{:}0.05{:}15$. What is the norm

of the difference between the two results? How do you compare the running times of the two methods?

(cf) Note that `Jkb(K,beta)` computes $J_k(\beta)$ of order $k = 1$:K, while the integration does for only $k =$ K.

```
function [J,JJ] = Jkb(K,beta) %the 1st kind of kth-order Bessel ftn
tmpk = ones(size(beta));
for k = 0:K
  tmp = tmpk;  JJ(k + 1,:) = tmp;
  for m = 1:100
    tmp = ?????????????????????;
    JJ(k + 1,:) = JJ(k + 1,:)+ tmp;
    if norm(tmp)<.001, break; end
    end
    tmpk = tmpk.*beta/2/(k + 1);
end
J = JJ(K+1,:);
```

```
%nm1p21b: Bessel_ftn
clear, clf
beta = 0:.05:15; K = 15;
tic
for i = 1:length(beta) %Integration
  J151(i) = quad('Bessel_integrand',0,pi,[],0,beta(i),K)/pi;
end
toc
tic, J152 = Jkb(K,beta); toc  %Recursive Computation
discrepancy = norm(J151-J152)
```

1.22 Find the four routines in Chapter 5 and 7, which are fabricated in a nested (recursive calling) structure.

(cf) Don't those algorithms, which are the souls of the routines, seem to have been born to be in a nested structure?

1.23 Avoiding Runtime Error in Case of Deficient/Nonadmissible Input Arguments

(a) Consider the MATLAB routine "`rotation_r(x,M)`", which you made in Problem 1.15(h). Does it work somehow when the user gives a negative integer as the second input argument M ? If not, add a statement so that it performs the rotation left by −M samples for M < 0, say, making

```
rotate_r([1 2 3 4 5],-2) = [3 4 5 1 2]
```

(b) Consider the routine '`trpzds(f,a,b,N)`' in Section 5.6, which computes the integral of function f over [a, b] by dividing the integration interval into N sections and applying the trapezoidal rule. If the user tries to use it without the fourth input argument N, will it work? If not, make it work with N $= 1000$ by default even without the fourth input argument N.

```
function INTf = trpzds(f,a,b,N)
%integral of f(x) over [a,b] by trapezoidal rule with N segments
if abs(b - a) < eps | N <= 0, INTf = 0; return; end
h = (b - a)/N; x = a+[0:N]*h;
fx = feval(f,x); %values of f for all nodes
INTf = h*((fx(1)+ fx(N + 1))/2 + sum(fx(2:N))); %Eq.(5.6.1)
```

1.24 Parameter Passing through `varargin`

Consider the integration routine '`trpzds(f,a,b,N)`' in Section 5.6. Can you apply it to compute the integral of a function with some parameter(s), like the '`Bessel_integrand(x,beta,k)`' that you defined in Problem 1.21? If not, modify it so that it works for a function with some parameter(s) (see Section 1.3.6) and save it in the M-file named '`trpzds_par.m`'. Then replace the '`quad()`' statement in the program '`nm1p21b`' (introduced in P1.21) by an appropriate '`trpzds_par()`' statement (with `N` = `1000`) and run the program. What is the discrepancy between the integration results obtained by this routine and the recursive computation based on Problem 1.21.4(b)? Is it comparable with that obtained with '`quad()`'? How do you compare the running time of this routine with that of '`quad()`'? Why do you think it takes so much time to execute the '`quad()`' routine?

1.25 Adaptive Input Argument to Avoid Runtime Error in the Case of Different Input Arguments

Consider the integration routine '`trpzds(f,a,b,N)`' in Section 5.6. If some user tries to use this routine with the following statement, will it work?

`trpzds(f,[a b],N)`   or   `trpzds(f,[a b])`

If not, modify it so that it works for such a usage (with a bound vector as the second input argument) as well as for the standard usage and save it in the M-file named '`trpzds_bnd.m`'. Then try it to find the intergal of e^{-t} for [0,100] by typing the following statements in the MATLAB command window. What did you get?

```
>>ftn=inline('exp(-t)','t');
>>trpzds_bnd(ftn,[0 100],1000)
>>trpzds_bnd(ftn,[0 100])
```

1.26 CtFT(Continuous-Time Fourier Transform) of an Arbitrary Signal

Consider the following definitions of CtFT and ICtFT(Inverse CtFT) [W-4]:

$$X(\omega) = F\{x(t)\} = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}dt\text{: CtFT} \qquad \text{(P1.26.1a)}$$

$$x(t) = F^{-1}\{X(\omega)\} = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)e^{j\omega t}d\omega\text{: ICtFT} \qquad \text{(P1.26.1b)}$$

(a) Similarly to the MATLAB routine "`CtFT1(x,Dt,w)`" computing the CtFT (P1.26.1a) of `x(t)` over `[-Dt,Dt ]` for `w`, make a MATLAB routine "`ICtFT1(X,Bw,t)`" computing the ICtFT (P1.26.1b) of `X(w)` over `[-Bw, Bw]` for `t`. You can choose whatever integral routine including '`trpzds_par()`' (Problem 1.24) and '`quad()`', considering the running time.

(b) The following program '`nm1p26`' finds the CtFT of a rectangular pulse (with duration $[-1,1]$) defined by '`rDt()`' for $\omega = [-6\pi, +6\pi]$ and the ICtFT of a sinc spectrum (with bandwidth 2π) defined by '`sincBw()`' for $t = [-5, +5]$. After having saved the routines into M-files with the appropriate names, run the program to see the rectangular pulse, its CtFT spectrum, a sinc spectrum, and its ICtFT. If it doesen't work, modify/supplement the routines so that you can rerun it to see the signals and their spectra.

```
function Xw = CtFT1(x,Dt,w)
x_ejkwt = inline([x '(t).*exp(-j*w*t)'],'t','w');
Xw = trpzds_par(x_ejkwt,-Dt,Dt,1000,w);
%Xw = quad(x_ejkwt,-Dt,Dt,[],0,w);
```

```
function xt = ICtFT1(X,Bw,t)
```

```
function x = rDt(t)

x = (-D/2 <= t & t <= D/2);
```

```
function X = sincBw(w)

X = 2*pi/B*sinc(w/B);
```

```
%nm1p26: CtFT and ICtFT
clear, clf
global B D
%CtFT of a Rectangular Pulse Function
t = [-50:50]/10; %time vector
w = [-60:60]/10*pi; %frequency vector
D = 1; %Duration of a rectangular pulse rD(t)
for k = 1:length(w), Xw(k) = CtFT1('rDt',D*5,w(k)); end
subplot(221), plot(t,rDt(t))
subplot(222), plot(w,abs(Xw))
%ICtFT of a Sinc Spectrum
B = 2*pi; %Bandwidth of a sinc spectrum sncB(w)
for n = 1:length(t), xt(n) = ICtFT1('sincBw',B*5,t(n));
end
subplot(223), plot(t,real(xt))
subplot(224), plot(w,sincBw(w))
```

2

SYSTEM OF LINEAR EQUATIONS

In this chapter, we deal with several numerical schemes for solving a system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ \cdots\cdots\cdots \quad &= \;. \\ a_{M1}x_1 + a_{M2}x_2 + \cdots + a_{MN}x_N &= b_M \end{aligned} \tag{2.0.1a}$$

which can be written in a compact form by using a matrix–vector notation as

$$A_{M\times N}\mathbf{x} = \mathbf{b} \tag{2.0.1b}$$

where

$$A_{M\times N} = \begin{bmatrix} a_{11} & a_{12} & \cdot & \cdot & a_{1N} \\ a_{21} & a_{22} & \cdot & \cdot & a_{2N} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{M1} & a_{M2} & \cdot & \cdot & a_{MN} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ x_N \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ b_M \end{bmatrix}$$

We will deal with the three cases:

(i) The case where the number (M) of equations and the number (N) of unknowns are equal ($M = N$) so that the coefficient matrix $A_{M\times N}$ is square.

(ii) The case where the number (M) of equations is smaller than the number (N) of unknowns ($M < N$) so that we might have to find the minimum-norm solution among the numerous solutions.

(iii) The case where the number of equations is greater than the number of unknowns ($M > N$) so that there might exist no exact solution and we must find a solution based on global error minimization, like the "LSE (Least-squares error) solution."

2.1 SOLUTION FOR A SYSTEM OF LINEAR EQUATIONS

2.1.1 The Nonsingular Case ($M = N$)

If the number (M) of equations and the number (N) of unknowns are equal ($M = N$), then the coefficient matrix A is square so that the solution can be written as

$$\mathbf{x} = A^{-1}\,\mathbf{b} \tag{2.1.1}$$

so long as the matrix A is not singular. There are MATLAB commands for this job.

```
>>A = [1 2;3 4]; b = [-1;-1];
>>x = A^-1*b %or, x = inv(A)*b
  x = 1.0000
     -1.0000
```

What if A is square, but singular?

```
>>A = [1 2;2 4]; b = [-1;-1];
>>x = A^-1*b

  Warning: Matrix is singular to working precision.
   x = -Inf
       -Inf
```

This is the case where some or all of the rows of the coefficient matrix A are dependent on other rows and so the rank of A is deficient, which implies that there are some equations equivalent to or inconsistent with other equations. If we remove the dependent rows until all the (remaining) rows are independent of each other so that A has full rank (equal to M), it leads to the case of $M < N$, which will be dealt with in the next section.

2.1.2 The Underdetermined Case ($M < N$): Minimum-Norm Solution

If the number (M) of equations is less than the number (N) of unknowns, the solution is not unique, but numerous. Suppose the M rows of the coefficient matrix A are independent. Then, any N-dimensional vector can be decomposed into two components

$$\mathbf{x} = \mathbf{x}^{+} + \mathbf{x}^{-} \tag{2.1.2}$$

where the one is in the row space $\mathcal{R}(A)$ of A that can be expressed as a linear combination of the M row vectors

$$\mathbf{x}^+ = A^T\boldsymbol{\alpha} \tag{2.1.3}$$

and the other is in the null space $\mathcal{N}(A)$ orthogonal(perpendicular) to the row space[1] so that

$$A\mathbf{x}^- = \mathbf{0} \tag{2.1.4}$$

Substituting the arbitrary N-dimensional vector representation (2.1.2) into Eq. (2.0.1) yields

$$A(\mathbf{x}^+ + \mathbf{x}^-) = AA^T\boldsymbol{\alpha} + A\mathbf{x}^- \overset{(2.1.4)}{=} AA^T\boldsymbol{\alpha} = \mathbf{b} \tag{2.1.5}$$

Since AA^T is supposedly a nonsingular $M \times M$ matrix resulting from multiplying an $M \times N$ matrix by an $N \times M$ matrix, we can solve this equation for $\boldsymbol{\alpha}$ to get

$$\boldsymbol{\alpha}^o = [AA^T]^{-1}\mathbf{b} \tag{2.1.6}$$

Then, substituting Eq. (2.1.6) into Eq. (2.1.3) yields

$$\mathbf{x}^{o+} \overset{(2.1.3)}{=} A^T\boldsymbol{\alpha}^o \overset{(2.1.6)}{=} A^T[AA^T]^{-1}\mathbf{b} \tag{2.1.7}$$

This satisfies Eq. (2.0.1) and thus qualifies as its solution. However, it is far from being a unique solution because the addition of any vector $\mathbf{x}^-$ (in the null space) satisfying Eq. (2.1.4) to $\mathbf{x}^{o+}$ still satisfies Eq. (2.0.1) [as seen from Eq. (2.1.5)], yielding infinitely many solutions.

Based on the principle that any one of the two perpendicular legs is shorter than the hypotenuse in a right-angled triangle, Eq. (2.1.7) is believed to represent the *minimum-norm solution*. Note that the matrix $A^T[AA^T]^{-1}$ is called the right pseudo- (generalized) inverse of A (see item 2 in Remark 1.1).

MATLAB has the `pinv()` command for obtaining the pseudo-inverse. We can use this command or the slash(/) operator to find the minimum-norm solution (2.1.7) to the system of linear equations (2.0.1).

```
>>A = [1 2]; b = 3;
>>x = pinv(A)*b %x = A'*(A*A')^ - 1*b or eye(size(A,2))/A*b, equivalently
   x = 0.6000
       1.2000
```

Remark 2.1. Projection Operator and Minimum-Norm Solution

1. The solution (2.1.7) can be viewed as the projection of an arbitrary solution $\mathbf{x}^o$ onto the row space $\mathcal{R}(A)$ of the coefficient matrix A spanned by the

[1] See the website @http://www.psc.edu/~burkardt/papers/linear_glossary.html

row vectors. The remaining component of the solution $\mathbf{x}^o$

$$\mathbf{x}^{o-} = \mathbf{x}^o - \mathbf{x}^{o+} = \mathbf{x}^o - A^T[AA^T]^{-1}\mathbf{b} = \mathbf{x}^o - A^T[AA^T]^{-1}A\mathbf{x}^o$$
$$= [I - A^T[AA^T]^{-1}A]\mathbf{x}^o$$

is in the null space $\mathcal{N}(A)$, since it satisfies Eq. (2.1.4). Note that

$$P_A = [I - A^T[AA^T]^{-1}A]$$

is called the projection operator.

2. The solution (2.1.7) can be obtained by applying the Lagrange multiplier method (Section 7.2.1) to the constrained optimization problem in which we must find a vector $\mathbf{x}$ minimizing the (squared) norm $\|\mathbf{x}\|^2$ subject to the equality constraint $A\mathbf{x} = \mathbf{b}$.

$$\text{Min } l(\mathbf{x}, \boldsymbol{\lambda}) \overset{\text{Eq.(7.2.2)}}{=} \frac{1}{2}\|\mathbf{x}\|^2 - \boldsymbol{\lambda}^T(A\mathbf{x} - \mathbf{b}) = \frac{1}{2}\mathbf{x}^T\mathbf{x} - \boldsymbol{\lambda}^T(A\mathbf{x} - \mathbf{b})$$

By using Eq. (7.2.3), we get

$$\frac{\partial}{\partial \mathbf{x}}J = \mathbf{x} - A^T\boldsymbol{\lambda} = \mathbf{0}; \qquad \mathbf{x} = A^T\boldsymbol{\lambda} = A^T[AA^T]^{-1}\mathbf{b}$$
$$\frac{\partial}{\partial \boldsymbol{\lambda}}J = A\mathbf{x} - \mathbf{b} = \mathbf{0}; \qquad AA^T\boldsymbol{\lambda} = \mathbf{b}; \qquad \boldsymbol{\lambda} = [AA^T]^{-1}\mathbf{b}$$

Example 2.1. Minimum-Norm Solution. Consider the problem of solving the equation

$$[1 \quad 2]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 3; \quad A\mathbf{x} = \mathbf{b}, \quad \text{where } A = [1 \quad 2], \quad \mathbf{b} = 3 \qquad \text{(E2.1.1)}$$

This has infinitely many solutions and any $\mathbf{x} = [x_1 \quad x_2]^T$ satisfying this equation, or, equivalently,

$$x_1 + 2x_2 = 3; \qquad x_2 = -\frac{1}{2}x_1 + \frac{3}{2} \qquad \text{(E2.1.2)}$$

is a qualified solution. Equation (E2.1.2) describes the solution space as depicted in Fig. 2.1.

On the other hand, any vector in the row space of the coefficient matrix A can be expressed by Eq. (2.1.3) as

$$\mathbf{x}^+ = A^T\,\alpha = \begin{bmatrix} 1 \\ 2 \end{bmatrix}\alpha \qquad (\alpha \text{ is a scalar, since } M = 1) \qquad \text{(E2.1.3)}$$

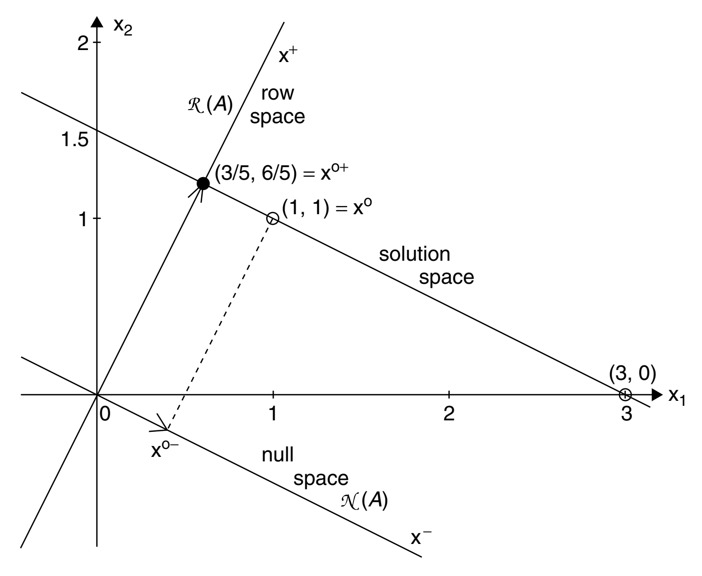


Figure 2.1 A minimum-norm solution.

and any vector in the null space of A can be expressed by Eq. (2.1.4) as

$$A\mathbf{x}^{-} = [\,1 \quad 2\,]\begin{bmatrix} x_1^{-} \\ x_2^{-} \end{bmatrix} = 0; \qquad x_2^{-} = -\frac{1}{2}x_1^{-} \tag{E2.1.4}$$

We use Eq. (2.1.7) to obtain the minimum-norm solution

$$\mathbf{x}^{o+} = A^T[AA^T]^{-1}\mathbf{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}\left([\,1 \quad 2\,]\begin{bmatrix} 1 \\ 2 \end{bmatrix}\right)^{-1} 3 = \frac{3}{5}\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 1.2 \end{bmatrix} \tag{E2.1.5}$$

Note from Fig. 2.1 that the minimum-norm solution $\mathbf{x}^{o+}$ is the intersection of the solution space and the row space and is the closest to the origin among the vectors in the solution space.

2.1.3 The Overdetermined Case ($M > N$): LSE Solution

If the number (M) of (independent) equations is greater than the number (N) of unknowns, there exists no solution satisfying all the equations strictly. Thus we try to find the LSE (least-squares error) solution minimizing the norm of the (inevitable) error vector

$$\mathbf{e} = A\mathbf{x} - \mathbf{b} \tag{2.1.8}$$

Then, our problem is to minimize the objective function

$$J = \tfrac{1}{2}\|\mathbf{e}\|^2 = \tfrac{1}{2}\|A\mathbf{x} - \mathbf{b}\|^2 = \tfrac{1}{2}[A\mathbf{x} - \mathbf{b}]^T[A\mathbf{x} - \mathbf{b}] \tag{2.1.9}$$

whose solution can be obtained by setting the derivative of this function (2.1.9) with respect to $\mathbf{x}$ to zero.

$$\frac{\partial}{\partial \mathbf{x}} J = A^T[A\mathbf{x} - \mathbf{b}] = \mathbf{0}; \qquad \mathbf{x}^o = [A^T A]^{-1} A^T \mathbf{b} \tag{2.1.10}$$

Note that the matrix A having the number of rows greater than the number of columns ($M > N$) does not have its inverse, but has its left pseudo (generalized) inverse $[A^T A]^{-1} A^T$ as long as A is not rank-deficient—that is, all of its columns are independent of each other (see item 2 in Remark 1.1). The left pseudo-inverse matrix can be computed by using the MATLAB command `pinv()`.

The LSE solution (2.1.10) can be obtained by using the `pinv()` command or the backslash (\) operator.

```
>>A = [1; 2]; b = [2.1; 3.9];
>>x = pinv(A)*b %A\b or x = (A'*A)^-1*A'*b, equivalently
  x = 1.9800
```

```
function  x = lin_eq(A,B)
%This function finds the solution to Ax = B
[M,N] = size(A);
if size(B,1) ~= M
  error('Incompatible dimension of A and B in lin_eq()!')
end
if M == N, x = A^-1*B; %x = inv(A)*B or gaussj(A,B); %Eq.(2.1.1)
 elseif M < N %Minimum-norm solution (2.1.7)
   x = pinv(A)*B; %A'*(A*A')^-1*B; or eye(size(A,2))/A*B
 else %LSE solution (2.1.10) for M > N
   x = pinv(A)*B; %(A'*A)^-1*A'*B or x = A\B
end
```

The above MATLAB routine `lin_eq()` is designed to solve a given set of equations, covering all of the three cases in Sections 2.1.1, 2.1.2, and 2.1.3.

(cf) The power of the `pinv()` command is beyond our imagination as you might have felt in Problem 1.14. Even in the case of $M < N$, it finds us a LS solution if the equations are inconsistent. Even in the case of $M > N$, it finds us a minimum-norm solution if the equations are redundant. Actually, the three cases can be dealt with by a single `pinv()` command in the above routine.

2.1.4 RLSE (Recursive Least-Squares Estimation)

In this section we will see the so-called RLSE (Recursive Least-Squares Estimation) algorithm, which is a recursive method to compute the LSE solution. Suppose we know the theoretical relationship between the temperature $t[^\circ]$ and

the resistance $R[\Omega]$ of a resistor as

$$c_1 t + c_2 = R$$

and we have lots of experimental data $\{(t_1, R_1), (t_2, R_2), \ldots, (t_k, R_k)\}$ collected up to time k. Since the above equation cannot be satisfied for all the data with any value of the parameters c_1 and c_2, we should try to get the parameter estimates that are optimal in some sense. This corresponds to the overdetermined case dealt with in the previous section and can be formulated as an LSE problem that we must solve a set of linear equations

$$A_k \mathbf{x}_k \approx \mathbf{b}_k, \quad \text{where } A_k = \begin{bmatrix} t_1 & 1 \\ t_2 & 1 \\ \cdot & \cdot \\ t_k & 1 \end{bmatrix}, \ \mathbf{x}_k = \begin{bmatrix} c_{1,k} \\ c_{2,k} \end{bmatrix}, \text{ and } \mathbf{b}_k = \begin{bmatrix} R_1 \\ R_2 \\ \cdot \\ R_k \end{bmatrix}$$

for which we can apply Eq. (2.1.10) to get the solution as

$$\mathbf{x}_k = [A_k^T A_k]^{-1} A_k^T \mathbf{b}_k \tag{2.1.11}$$

Now, we are given a new experimental data (t_{k+1}, R_{k+1}) and must find the new parameter estimate

$$\mathbf{x}_{k+1} = [A_{k+1}^T A_{k+1}]^{-1} A_{k+1}^T \mathbf{b}_{k+1} \tag{2.1.12}$$

with

$$A_{k+1} = \begin{bmatrix} t_1 & 1 \\ \cdot & \cdot \\ t_k & 1 \\ t_{k+1} & 1 \end{bmatrix}, \quad \mathbf{x}_{k+1} = \begin{bmatrix} c_{1,k+1} \\ c_{2,k+1} \end{bmatrix}, \quad \text{and} \quad \mathbf{b}_{k+1} = \begin{bmatrix} R_1 \\ \cdot \\ R_k \\ R_{k+1} \end{bmatrix}$$

How do we compute this? If we discard the previous estimate $\mathbf{x}_k$ and make direct use of Eq. (2.1.12) to compute the next estimate $\mathbf{x}_{k+1}$ every time a new data pair is available, the size of matrix A will get bigger and bigger as the data pile up, eventually defying any powerful computer in this world.

How about updating the previous estimate by just adding the correction term based on the new data to get the new estimate? This is the basic idea of the RLSE algorithm, which we are going to trace and try to understand. In order to do so, let us define the notations

$$A_{k+1} = \begin{bmatrix} A_k \\ \mathbf{a}_{k+1}^T \end{bmatrix}, \quad \mathbf{a}_{k+1} = \begin{bmatrix} t_{k+1} \\ 1 \end{bmatrix}, \quad \mathbf{b}_{k+1} = \begin{bmatrix} \mathbf{b}_k \\ R_{k+1} \end{bmatrix}, \quad \text{and} \quad P_k = [A_k^T A_k]^{-1} \tag{2.1.13}$$

and see how the inverse matrix P_k is to be updated on arrival of the new data (t_{k+1}, R_{k+1}).

$$P_{k+1} = [A_{k+1}^T A_{k+1}]^{-1} = \left[[\, A_k^T \quad \mathbf{a}_{k+1} \,] \begin{bmatrix} A_k \\ \mathbf{a}_{k+1}^T \end{bmatrix} \right]^{-1}$$

$$= [A_k^T A_k + \mathbf{a}_{k+1}\mathbf{a}_{k+1}^T]^{-1} = [P_k^{-1} + \mathbf{a}_{k+1}\mathbf{a}_{k+1}^T]^{-1} \tag{2.1.14}$$

(Matrix Inversion Lemma in Appendix B)

$$P_{k+1} = P_k - P_k\mathbf{a}_{k+1}[\mathbf{a}_{k+1}^T P_k \mathbf{a}_{k+1} + 1]^{-1}\mathbf{a}_{k+1}^T P_k \tag{2.1.15}$$

It is interesting that $[\mathbf{a}_{k+1}^T P_k \mathbf{a}_{k+1} + 1]$ is nothing but a scalar and so we do not need to compute the matrix inverse thanks to the Matrix Inversion Lemma (Appendix B). It is much better in the computational aspect to use the recursive formula (2.1.15) than to compute $[A_{k+1}^T A_{k+1}]^{-1}$ directly. We can also write Eq. (2.1.12) in a recursive form as

$$\mathbf{x}_{k+1} \overset{(2.1.12,\,14)}{=} P_{k+1}A_{k+1}^T\mathbf{b}_{k+1} \overset{(2.1.13)}{=} P_{k+1}[A_k^T \;\; \mathbf{a}_{k+1}] \begin{bmatrix} \mathbf{b}_k \\ R_{k+1} \end{bmatrix}$$

$$= P_{k+1}[A_k^T\mathbf{b}_k + \mathbf{a}_{k+1}R_{k+1}] \overset{(2.1.11)}{=} P_{k+1}[A_k^T A_k\mathbf{x}_k + \mathbf{a}_{k+1}R_{k+1}]$$

$$\overset{(2.1.13)}{=} P_{k+1}[(A_{k+1}^T A_{k+1} - \mathbf{a}_{k+1}\mathbf{a}_{k+1}^T)\mathbf{x}_k + \mathbf{a}_{k+1}R_{k+1}]$$

$$\overset{(2.1.13)}{=} P_{k+1}[P_{k+1}^{-1}\mathbf{x}_k - \mathbf{a}_{k+1}\mathbf{a}_{k+1}^T\mathbf{x}_k + \mathbf{a}_{k+1}R_{k+1}]$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + P_{k+1}\mathbf{a}_{k+1}(R_{k+1} - \mathbf{a}_{k+1}^T\mathbf{x}_k) \tag{2.1.16}$$

We can use Eq. (2.1.15) to rewrite the gain matrix $P_{k+1}\mathbf{a}_{k+1}$ premultiplied by the ‘error’ to make the correction term on the right-hand side of Eq. (2.1.16) as

$$K_{k+1} = P_{k+1}\mathbf{a}_{k+1} \overset{(2.1.15)}{=} [P_k - P_k\mathbf{a}_{k+1}[\mathbf{a}_{k+1}^T P_k\mathbf{a}_{k+1} + 1]^{-1}\mathbf{a}_{k+1}^T P_k]\mathbf{a}_{k+1}$$

$$= P_k\mathbf{a}_{k+1}[I - [\mathbf{a}_{k+1}^T P_k\mathbf{a}_{k+1} + 1]^{-1}\mathbf{a}_{k+1}^T P_k\mathbf{a}_{k+1}]$$

$$= P_k\mathbf{a}_{k+1}[\mathbf{a}_{k+1}^T P_k\mathbf{a}_{k+1} + 1]^{-1}\{[\mathbf{a}_{k+1}^T P_k\mathbf{a}_{k+1} + 1] - \mathbf{a}_{k+1}^T P_k\mathbf{a}_{k+1}\}$$

$$K_{k+1} = P_k\mathbf{a}_{k+1}[\mathbf{a}_{k+1}^T P_k\mathbf{a}_{k+1} + 1]^{-1} \tag{2.1.17}$$

and substitute this back into Eq. (2.1.15) to write it as

$$P_{k+1} = P_k - K_{k+1}\mathbf{a}_{k+1}^T P_k \tag{2.1.18}$$

The following MATLAB routine “`rlse_online()`” implements this RLSE (Recursive Least-Squares Estimation) algorithm that updates the parameter estimates by using Eqs. (2.1.17), (2.1.16), and (2.1.18). The MATLAB program

"do_rlse.m" updates the parameter estimates every time new data arrive and compares the results of the on-line processing with those obtained by the off-line (batch job) processing—that is, by using Eq.(2.1.12) directly. Noting that

- the matrix $[A_k^T A_k]$ as well as $\mathbf{b}_k$ consists of information and is a kind of squared matrix that is nonnegative, and
- $[A_k^T A_k]$ will get larger, or, equivalently, $P_k = [A_k^T A_k]^{-1}$ will get smaller and, consequently, the gain matrix K_k will get smaller as valuable information data accumulate,

one could understand that P_k is initialized to a very large identity matrix, since no information is available in the beginning. Since a large/small P_k makes the correction term on the right-hand side of Eq. (2.1.16) large/small, the RLSE algorithm becomes more conservative and reluctant to learn from the new data as the data pile up, while it is willing to make use of the new data for updating the estimates when it is hungry for information in the beginning.

```
function [x,K,P] = rlse_online(aT_k1,b_k1,x,P)
K = P*aT_k1'/(aT_k1*P*aT_k1'+1); %Eq.(2.1.17)
x = x +K*(b_k1-aT_k1*x); %Eq.(2.1.16)
P = P-K*aT_k1*P;  %Eq.(2.1.18)
```

```
%do_rlse
clear
xo = [2  1]'; %The true value of unknown coefficient vector
NA = length(xo);
x = zeros(NA,1); P = 100*eye(NA,NA);
for k = 1:100
   A(k,:) = [k*0.01 1];
   b(k,:) = A(k,:)*xo +0.2*rand;
   [x,K,P] = rlse_online(A(k,:),b(k,:),x,P);
 end
 x   % the final parameter estimate
 A\b % for comparison with the off-line processing (batch job)
```

2.2 SOLVING A SYSTEM OF LINEAR EQUATIONS

2.2.16 Gauss Elimination

For simplicity, we assume that the coefficient matrix A in Eq. (2.0.1) is a non-singular 3×3 matrix with $M = N = 3$. Then we can write the equation as

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \tag{2.2.0a}$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \tag{2.2.0b}$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \tag{2.2.0c}$$

First, to remove the x_1 terms from equations (2.2.0.m) other than (2.2.0.a), we subtract (2.2.0a)$\times a_{m1}/a_{11}$ from each of them to get

$$a_{11}^{(0)}x_1 + a_{12}^{(0)}x_2 + a_{13}^{(0)}x_3 = b_1^{(0)} \tag{2.2.1a}$$

$$a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 = b_2^{(1)} \tag{2.2.1b}$$

$$a_{32}^{(1)}x_2 + a_{33}^{(1)}x_3 = b_3^{(1)} \tag{2.2.1c}$$

with

$$a_{mn}^{(0)} = a_{mn}, \quad b_m^{(0)} = b_m \qquad \text{for } m, n = 1, 2, 3 \tag{2.2.2a}$$

$$a_{mn}^{(1)} = a_{mn}^{(0)} - (a_{m1}^{(0)}/a_{11}^{(0)})a_{1n}^{(0)}, \quad b_m^{(1)} = b_m^{(0)} - (a_{m1}^{(0)}/a_{11}^{(0)})b_1^{(0)} \qquad \text{for } m, n = 2, 3 \tag{2.2.2b}$$

We call this work 'pivoting at a_{11}' and call the center element a_{11} a 'pivot'.

Next, to remove the x_2 term from Eq. (2.2.1c) other than (2.2.1a,b), we subtract (2.2.1b)$\times a_{m2}^{(1)}/a_{22}^{(1)}(m = 3)$ from it to get

$$a_{11}^{(0)}x_1 + a_{12}^{(0)}x_2 + a_{13}^{(0)}x_3 = b_1^{(0)} \tag{2.2.3a}$$

$$a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 = b_2^{(1)} \tag{2.2.3b}$$

$$a_{33}^{(2)}x_3 = b_3^{(2)} \tag{2.2.3c}$$

with

$$a_{mn}^{(2)} = a_{mn}^{(1)} - (a_{m2}^{(1)}/a_{22}^{(1)})a_{2n}^{(1)}, \quad b_m^{(2)} = b_m^{(1)} - (a_{m2}^{(1)}/a_{22}^{(1)})b_2^{(1)} \qquad \text{for } m, n = 3 \tag{2.2.4}$$

We call this procedure 'Gauss forward elimination' and can generalize the updating formula (2.2.2)/(2.2.4) as

$$a_{mn}^{(k)} = a_{mn}^{(k-1)} - (a_{mk}^{(k-1)}/a_{kk}^{(k-1)})a_{kn}^{(k-1)} \qquad \text{for } m, n = k+1, k+2, \ldots, M \tag{2.2.5a}$$

$$b_m^{(k)} = b_m^{(k-1)} - (a_{mk}^{(k-1)}/a_{kk}^{(k-1)})b_k^{(k-1)} \qquad \text{for } m = k+1, k+2, \ldots, M \tag{2.2.5b}$$

After having the triangular matrix–vector equation as Eq. (2.2.3), we can solve Eq. (2.2.3c) first to get

$$x_3 = b_3^{(2)}/a_{33}^{(2)} \tag{2.2.6a}$$

and then substitute this result into Eq. (2.2.3b) to get

$$x_2 = (b_2^{(1)} - a_{23}^{(1)}x_3)/a_{22}^{(1)} \tag{2.2.6b}$$

Successively, we substitute Eqs. (2.2.6a,b) into Eq.(2.2.3a) to get

$$x_1 = \left(b_1^{(0)} - \sum_{n=2}^{3} a_{1n}^{(0)}x_n\right)/a_{11}^{(0)} \tag{2.2.6c}$$

We call this procedure 'backward substitution' and can generalize the solution formula (2.2.6) as

$$x_m = \left(b_m^{(m-1)} - \sum_{n=m+1}^{M} a_{mn}^{(m-1)} x_n \right) / a_{mm}^{(m-1)} \qquad \text{for } m = M, M-1, \ldots, 1 \tag{2.2.7}$$

In this way, the Gauss elimination procedure consists of two steps, namely, forward elimination and backward substitution. Noting that

- this procedure has nothing to do with the specific values of the unknown variable x_m's and involves only the coefficients, and
- the formulas (2.2.5a) on the coefficient matrix A and (2.2.5b) on the RHS (right-hand side) vector **b** conform with each other,

we will augment A with **b** and put the formulas (2.2.5a,b) together into one framework when programming the Gauss forward elimination procedure.

2.2.2 Partial Pivoting

The core formula (2.2.5) used for Gauss elimination requires division by $a_{kk}^{(k-1)}$ at the kth stage, where $a_{kk}^{(k-1)}$ is the diagonal element in the kth row. What if $a_{kk}^{(k-1)} = 0$? In such a case, it is customary to switch the kth row and another row below it having the element of the largest absolute value in the kth column. This procedure, called 'partial pivoting', is recommended for reducing the round-off error even in the case where the kth pivot $a_{kk}^{(k-1)}$ is not zero.

Let us consider the following example:

$$\begin{bmatrix} 0 & 1 & 1 \\ 2 & -1 & -1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 = 2 \\ b_2 = 0 \\ b_3 = 1 \end{bmatrix} \tag{2.2.8}$$

We construct the augmented matrix by combining the coefficient matrix and the RHS vector to write

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 2 & -1 & -1 & 0 \\ 1 & 1 & -1 & 1 \end{bmatrix} \begin{matrix} : & r_1 \\ : & r_2 \\ : & r_3 \end{matrix} \tag{2.2.9}$$

and apply the Gauss elimination procedure.

In the stage of forward elimination, we want to do pivoting at a_{11}, but a_{11} cannot be used as the pivoting element because it is zero. So we switch the first row and the second row having the element of the largest absolute value in the first column.

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & b_1^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} & b_2^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} & b_3^{(1)} \end{bmatrix} = \begin{bmatrix} 2 & -1 & -1 & 0 \\ 0 & 1 & 1 & 2 \\ 1 & 1 & -1 & 1 \end{bmatrix} \begin{matrix} : & r_1^{(1)} \\ : & r_2^{(1)} \\ : & r_3^{(1)} \end{matrix} \tag{2.2.10a}$$

Then we do pivoting at $a_{11}^{(1)}$ by applying Eq. (2.2.2) to get

$$\begin{matrix} r_1^{(1)} \\ r_2^{(1)} - a_{21}^{(1)}/a_{11}^{(1)} \times r_1^{(1)} \\ r_3^{(1)} - a_{31}^{(1)}/a_{11}^{(1)} \times r_1^{(1)} \end{matrix} \begin{matrix} \rightarrow \\ \rightarrow \\ \rightarrow \end{matrix} \begin{bmatrix} a_{11}^{(2)} & a_{12}^{(2)} & a_{13}^{(2)} & b_1^{(2)} \\ a_{21}^{(2)} & a_{22}^{(2)} & a_{23}^{(2)} & b_2^{(2)} \\ a_{31}^{(2)} & a_{32}^{(2)} & a_{33}^{(2)} & b_3^{(2)} \end{bmatrix}$$

$$= \begin{bmatrix} 2 & -1 & -1 & 0 \\ 0 & 1 & 1 & 2 \\ 0 & 3/2 & -1/2 & 1 \end{bmatrix} \begin{matrix} : & r_1^{(2)} \\ : & r_2^{(2)} \\ : & r_3^{(2)} \end{matrix} \tag{2.2.10b}$$

Here, instead of pivoting at $a_{22}^{(2)}$, we switch the second row and the third row having the element of the largest absolute value among the elements not above $a_{22}^{(2)}$ in the second column.

$$\begin{bmatrix} a_{11}^{(3)} & a_{12}^{(3)} & a_{13}^{(3)} & b_1^{(3)} \\ a_{21}^{(3)} & a_{22}^{(3)} & a_{23}^{(3)} & b_2^{(3)} \\ a_{31}^{(3)} & a_{32}^{(3)} & a_{33}^{(3)} & b_3^{(3)} \end{bmatrix} = \begin{bmatrix} 2 & -1 & -1 & 0 \\ 0 & 3/2 & -1/2 & 1 \\ 0 & 1 & 1 & 2 \end{bmatrix} \begin{matrix} : & r_1^{(3)} \\ : & r_2^{(3)} \\ : & r_3^{(3)} \end{matrix} \tag{2.2.10c}$$

And we do pivoting at $a_{22}^{(3)}$ by applying Eq. (2.2.4)—more generally, Eq. (2.2.5)—to get the upper-triangularized form:

$$\begin{matrix} r_1^{(3)} \\ r_2^{(3)} \\ r_3^{(3)} - a_{31}^{(3)}/a_{11}^{(3)} \times r_2^{(3)} \end{matrix} \begin{matrix} \rightarrow \\ \rightarrow \\ \rightarrow \end{matrix} \begin{bmatrix} a_{11}^{(4)} & a_{12}^{(4)} & a_{13}^{(4)} & b_1^{(4)} \\ a_{21}^{(4)} & a_{22}^{(4)} & a_{23}^{(4)} & b_2^{(4)} \\ a_{31}^{(4)} & a_{32}^{(4)} & a_{33}^{(4)} & b_3^{(4)} \end{bmatrix}$$

$$= \begin{bmatrix} 2 & -1 & -1 & 0 \\ 0 & 3/2 & -1/2 & 1 \\ 0 & 0 & 4/3 & 4/3 \end{bmatrix} \begin{matrix} : & r_1^{(4)} \\ : & r_2^{(4)} \\ : & r_3^{(4)} \end{matrix} \tag{2.2.10d}$$

Now, in the stage of backward substitution, we apply Eq. (2.2.6), more generally, Eq. (2.2.7) to get the final solution as

$$\begin{aligned} x_3 &= b_3^{(4)}/a_{33}^{(4)} = (4/3)/(4/3) = 1 \\ x_2 &= (b_2^{(4)} - a_{23}^{(4)} x_3)/a_{22}^{(4)} = (1 - (-1/2) \times 1)/(3/2) = 1 \\ x_1 &= \left(b_1^{(4)} - \sum_{n=2}^{3} a_{1n}^{(4)} x_n \right) / a_{11}^{(4)} = (0 - (-1) \times 1 - (-1) \times 1)/2 = 1 \end{aligned} \tag{2.2.11}$$

$$[x_1 \quad x_2 \quad x_3] = [1 \quad 1 \quad 1] \tag{2.2.12}$$

Let us consider another system of equations.

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 = 2 \\ b_2 = 3 \\ b_3 = 1 \end{bmatrix} \tag{2.2.13}$$

We construct the augmented matrix by combining the coefficient matrix and the RHS vector to write

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 2 \\ 1 & 1 & 1 & 3 \\ 1 & -1 & 1 & 1 \end{bmatrix} \begin{matrix} : & r_1 \\ : & r_2 \\ : & r_3 \end{matrix} \tag{2.2.14}$$

and apply the Gauss elimination procedure.

First, noting that all the elements in the first column have the same absolute value and so we don't need to switch the rows, we do pivoting at a_{11}.

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & b_1^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} & b_2^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} & b_3^{(1)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & -1 & 0 & -1 \end{bmatrix} \begin{matrix} : & r_1^{(1)} \\ : & r_2^{(1)} \\ : & r_3^{(1)} \end{matrix} \tag{2.2.15a}$$

Second, without having to switch the rows, we perform pivoting at $a_{22}^{(1)}$.

$$\begin{matrix} r_1^{(1)} & \to \\ r_2^{(1)} & \to \\ r_3^{(1)} - a_{32}^{(1)}/a_{22}^{(1)} \times r_2^{(1)} & \to \end{matrix} \begin{bmatrix} a_{11}^{(2)} & a_{12}^{(2)} & a_{13}^{(2)} & b_1^{(2)} \\ a_{21}^{(2)} & a_{22}^{(2)} & a_{23}^{(2)} & b_2^{(2)} \\ a_{31}^{(2)} & a_{32}^{(2)} & a_{33}^{(2)} & b_3^{(2)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} : & r_1^{(2)} \\ : & r_2^{(2)} \\ : & r_3^{(2)} \end{matrix} \tag{2.2.15b}$$

Now, we are at the stage of backward substitution, but $a_{33}^{(2)}$, which is supposed to be the denominator in Eq. (2.2.7), is zero. We may face such a weird situation of zero division even during the forward elimination process where the pivot is zero; besides, we cannot find any (nonzero) element below it in the same column and on its right in the same row except the RHS element. In this case, we cannot go further. This implies that some or all rows of coefficient matrix A are dependent on others, corresponding to the case of redundancy (infinitely many solutions) or inconsistency (no exact solution). Noting that the RHS element of the zero row in Eq. (2.2.15.2) is also zero, we should declare the case of redundancy and may have to be satisfied with one of the infinitely many solutions being the RHS vector as

$$[x_1 \quad x_2 \quad x_3] = [b_1^{(2)} \quad b_2^{(2)} \quad b_3^{(2)}] = [2 \quad 1 \quad 0] \tag{2.2.16}$$

Furthermore, if we remove the all-zero row(s), the problem can be treated as an underdetermined case handled in Section 2.1.2. Note that, if the RHS element were not zero, we would have to declare the case of inconsistency, as will be illustrated.

Suppose that $b_1 = 1$ in Eq. (2.2.14). Then, the Gauss elimination would have proceeded as follows:

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 3 \\ 1 & -1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & -1 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 2 \end{bmatrix} \tag{2.2.17}$$

This ended up with an all-zero row except the nonzero RHS element, corresponding to the case of inconsistency. So we must declare the case of 'no exact solution' for this problem.

The following MATLAB routine "`gauss()`" implements the Gauss elimination algorithm, and the program "`do_gauss`" is designed to solve Eq. (2.2.8) by using "`gauss()`". Note that at every pivoting operation in the routine "`gauss()`", the pivot row is divided by the pivot element so that every diagonal element becomes one and that we don't need to perform any computation for the kth column at the kth stage, since the column is supposed to be all zeros but the kth element $a_{kk}^{(k)} = 1$.

```
function  x = gauss(A,B)
%The sizes of matrices A,B are supposed to be NA x NA and NA x NB.
%This function solves Ax = B by Gauss elimination algorithm.
NA = size(A,2);  [NB1,NB] = size(B);
if NB1 ~= NA, error('A and B must have compatible dimensions'); end
N = NA + NB; AB = [A(1:NA,1:NA) B(1:NA,1:NB)]; % Augmented matrix
epss = eps*ones(NA,1);
for k = 1:NA
   %Scaled Partial Pivoting at AB(k,k) by Eq.(2.2.20)
   [akx,kx] = max(abs(AB(k:NA,k))./ ...
              max(abs([AB(k:NA,k + 1:NA) epss(1:NA - k + 1)]'))');
   if akx < eps, error('Singular matrix and No unique solution'); end
   mx = k + kx - 1;
   if kx > 1 % Row change if necessary
      tmp_row = AB(k,k:N);
      AB(k,k:N) = AB(mx,k:N);
      AB(mx,k:N) = tmp_row;
   end
   % Gauss forward elimination
   AB(k,k + 1:N) = AB(k,k+1:N)/AB(k,k);
   AB(k,k) = 1;  %make each diagonal element one
   for m = k + 1: NA
      AB(m,k+1:N) = AB(m,k+1:N) - AB(m,k)*AB(k,k+1:N); %Eq.(2.2.5)
      AB(m,k) = 0;
   end
end
%backward substitution for a upper-triangular matrix eqation
% having all the diagonal elements equal to one
x(NA,:) = AB(NA,NA+1:N);
for m = NA-1: -1:1
   x(m,:) = AB(m,NA + 1:N)-AB(m,m + 1:NA)*x(m + 1:NA,:); %Eq.(2.2.7)
end
```

```
%do_gauss
A = [0 1 1;2 -1 -1;1 1 -1]; b = [2 0 1]'; %Eq.(2.2.8)
x = gauss(A,b)
x1 = A\b %for comparison with the result of backslash operation
```

(cf) The number of floating-point multiplications required in this routine '`gauss()`' is

$$\sum_{k=1}^{NA}\{(NA-k+1)(NA+NB-k)+NA-k+1\}+\sum_{k=1}^{NA-1}(NA-k)NB$$
$$=\sum_{k=1}^{NA}k(k+NB-1)-NB\sum_{k=1}^{NA}k+\sum_{k=1}^{NA}NA\cdot NB$$
$$=\frac{1}{6}(NA+1)NA(2NA+1)-\frac{1}{2}NA(NA+1)+NA^2NB$$
$$=\frac{1}{3}NA(NA+1)(NA-1)+NA^2NB$$
$$\approx\frac{1}{3}NA^3 \qquad \text{for } NA \succ NB \tag{2.2.18}$$

where NA is the size of the matrix A, and NB is the column dimension of the RHS matrix B.

Here are several things to note.

Remark 2.2. Partial Pivoting and Undetermined/Inconsistent Case

1. In Gauss or Gauss–Jordan elimination, some row switching is performed to avoid the zero division. Even without that purpose, it may be helpful for reducing the round-off error to fix

$$\text{Max}\{|a_{mk}|, k \le m \le M\} \tag{2.2.19}$$

as the pivot element in the kth iteration through some row switching, which is called 'partial pivoting.' Actually, it might be better off to fix

$$\text{Max}\left\{\frac{|a_{mk}|}{\text{Max}\{|a_{mn}|, k \le n \le M\}}, k \le m \le M\right\} \tag{2.2.20}$$

as the pivot element in the kth iteration, which is called 'scaled partial pivoting' or to do column switching as well as row switching for choosing the best (largest) pivot element, which is called 'full pivoting.' Note that if the columns are switched, the order of the unknown variables should be interchanged accordingly.

2. What if some diagonal element a_{kk} and all the elements below it in the same column are zero and, besides, all the elements in the row including a_{kk} are also zero except the RHS element? It implies that some or all rows of the coefficient matrix A are dependent on others, corresponding to the case of redundancy (infinitely many solutions) or inconsistency (no

exact solution). If even the RHS element is zero, it should be declared to be the case of redundancy. In this case, we can get rid of the all-zero row(s) and then treat the problem as the underdetermined case handled in Section 2.1.2. If the RHS element is only one nonzero in the row, it should be declared to be the case of inconsistency.

Example 2.2. Delicacy of Partial Pivoting. To get an actual feeling about the delicacy of partial pivoting, consider the following systems of linear equations, which apparently have $\mathbf{x}^o = [1 \quad 1]^T$ as their solutions.

(a) $$\mathrm{A}_1\mathbf{x} = \mathbf{b}_1 \quad \text{with } \mathrm{A}_1 = \begin{bmatrix} 10^{-15} & 1 \\ 1 & 10^{11} \end{bmatrix}, \quad \mathbf{b}_1 = \begin{bmatrix} 1 + 10^{-15} \\ 10^{11} + 1 \end{bmatrix} \tag{E2.2.1}$$

Without any row switching, the Gauss elimination procedure will find us the true solution only if there is no quantization error.

$$[A_1 \ \mathbf{b}_1] = \begin{bmatrix} 10^{-15} & 1 & 1 + 10^{-15} \\ 1 & 10^{11} & 10^{11} + 1 \end{bmatrix}$$

$$\xrightarrow[\text{elimination}]{\text{forward}} \begin{bmatrix} 1 & 10^{15} & 10^{15} + 1 \\ 0 & 10^{11} - 10^{15} & 10^{11} - 10^{15} \end{bmatrix} \xrightarrow[\text{substitution}]{\text{backward}} \mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

But, because of the round-off error, it will deviate from the true solution.

$$\xrightarrow[\text{elimination}]{\text{forward}} \begin{bmatrix} 1 & 10^{15} = 9.999999999999999\text{e+}014 & 10^{15} + 1 = 1.000000000000001\text{e+}015 \\ 0 & 10^{11} - 10^{15} & 10^{11} + 1 - (10^{15} - 1) \\ & = -9.998999999999999\text{e+}014 & = -9.999000000000000\text{e+}014 \end{bmatrix}$$

....

$$\xrightarrow[\text{substitution}]{\text{backward}} \mathbf{x} = \begin{bmatrix} 8.750000000000000\text{e-}001 \\ 1.000000000000000\text{e+}000 \end{bmatrix}$$

If we enforce the strategy of partial pivoting or scaled partial pivoting, the Gauss elimination procedure will give us much better result as follows:

$$[A_1 \ \mathbf{b}_1] \xrightarrow{\text{row swap}} \begin{bmatrix} 1 & 10^{11} & 10^{11} + 1 \\ 10^{-15} & 1 & 1 + 10^{-15} \end{bmatrix}$$

$$\xrightarrow[\text{elimination}]{\text{forward}} \begin{bmatrix} 1 & 10^{11} = 1.000\text{e+}011 & 10^{11} + 1 = 1.000000000010000\text{e+}011 \\ 0 & 1 - 10^{-4} = 9.999\text{e-}001 & 9.999000000000001\text{e-}001 \end{bmatrix}$$

....

$$\xrightarrow[\text{substitution}]{\text{backward}} \mathbf{x} = \begin{bmatrix} 9.999847412109375\text{e-}001 \\ 1.000000000000000\text{e+}000 \end{bmatrix}$$

(b) $$\mathrm{A}_2\,\mathbf{x} = \mathbf{b}_2 \quad \text{with } \mathrm{A}_2 = \begin{bmatrix} 10^{-14.6} & 1 \\ 1 & 10^{15} \end{bmatrix}, \quad \mathbf{b}_2 = \begin{bmatrix} 1 + 10^{-14.6} \\ 10^{15} + 1 \end{bmatrix} \tag{E2.2.2}$$

Without partial pivoting, the Gauss elimination procedure will give us a quite good result.

$$[A_1\ \mathbf{b}_1] = \begin{bmatrix} 1 & 10^{14.6} = 3.981071705534969\text{e+}014 & 10^{14.6}+1 = 3.981071705534979\text{e+}014 \\ 0 & 6.018928294465030\text{e+}014 & 6.018928294465030\text{e+}014 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 1 & 3.981071705534969\text{e+}014 & 3.981071705534979\text{e+}014 \\ 0 & 1 & 1 \end{bmatrix}$$

$$\xrightarrow[\text{substitution}]{\text{backward}} \mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

But, if we exchange the first row with the second row having the larger element in the first column according to the strategy of partial pivoting, the Gauss elimination procedure will give us a rather surprisingly bad result as follows:

$$\xrightarrow[\text{forward elimination}]{\text{row swapping}} \begin{bmatrix} 1 & 10^{15} = 1.000000000000000\text{e+}015 & 10^{15}+1 = 1.000000000000001\text{e+}015 \\ 0 & 1 - 10^{15}\cdot 10^{-14.6} & 1 + 10^{-14.6} - (1+10^{15})\cdot 10^{-14.6} \\ & = -1.5118864315095819 & = -1.5118864315095821 \end{bmatrix}$$

$$\xrightarrow[\text{substitution}]{\text{backward}} \mathbf{x} = \begin{bmatrix} 0.7500000000000000 \\ 1.0000000000000002 \end{bmatrix}$$

One might be happy to have the scaled partial pivoting scheme [Eq. (2.2.20)], which does not switch the rows in this case, since the relative magnitude (dominancy) of a_{11} in the first row is greater than that of a_{21} in the second row, that is, $10^{-14.6}/1 > 1/10^{15}$.

(c) $$\mathrm{A}_3\mathbf{x} = \mathbf{b}_3 \quad \text{with } \mathrm{A}_3 = \begin{bmatrix} 10^{15} & 1 \\ 1 & 10^{-14.6} \end{bmatrix}, \ \mathbf{b}_3 = \begin{bmatrix} 10^{15}+1 \\ 1+10^{-14.6} \end{bmatrix} \tag{E2.2.3}$$

With any pivoting scheme, we don't need to switch the rows, since the relative magnitude as well as the absolute magnitude of a_{11} in the first row is greater than those of a_{21} in the second row. Thus, the Gauss elimination procedure will go as follows:

$$\xrightarrow[\text{elimination}]{\text{forward}} \begin{bmatrix} 1 & 1.000000000000000\text{e-}015 & 1.000000000000001\text{e+}000 \\ 0 & 1.511886431509582\text{e-}015 & 1.332267629550188\text{e-}015 \end{bmatrix}$$

$$\xrightarrow[\text{substitution}]{\text{backward}} \mathbf{x} = \begin{bmatrix} 1.000000000000000 \\ 0.811955724875121 \end{bmatrix}$$

(cf) Note that the coefficient matrix, A_3 is the same as would be obtained by applying the full pivoting scheme for A_2 to have the largest pivot element. This example implies that the Gauss elimination with full pivoting scheme may produce a worse result than would be obtained with scaled partial pivoting scheme. As a matter of

factor, we cannot say that some pivoting scheme always yields better solution than other pivoting schemes, because the result depends on the random round-off error as well as the pivoting scheme (see Problem 2.2). But, in most cases, the scaled partial pivoting shows a reasonably good performance and that is why we adopt it in our routine "`gauss()`".

Remark 2.3. Computing Error, Singularity, and Ill-Condition

1. As the size of the matrix grows, the round-off errors are apt to accumulate and propagated in matrix operations to such a degree that zero may appear to be an absolutely small number, or a nonzero number very close to zero may appear to be zero. Therefore, it is not so simple a task to determine whether a zero or a number very close to zero is a real zero or not.
2. It is desirable, but not so easy, for us to discern the case of singularity from the case of ill-condition and to distinguish the case of redundancy from the case of inconsistency. In order to be able to give such a qualitative judgment in the right way based on some quantitative analysis, we should be equipped with theoretical knowledge as well as practical experience.
3. There are several criteria by which we judge the degree of ill-condition, such as how discrepant AA^{-1} is with the identity matrix, how far $\det\{A\}\det\{A^{-1}\}$ stays away from one(1), and so on:

$$AA^{-1} \stackrel{?}{=} I, \qquad [A^{-1}]^{-1} \stackrel{?}{=} A, \qquad \det(A)\det(A^{-1}) \stackrel{?}{=} 1 \tag{2.2.21}$$

   The MATLAB command `cond()` tells us the degree of ill-condition for a given matrix by the size of the condition number, which is defined as

$$cond(A) = \|A\|\|A^{-1}\| \qquad \text{with } \|A\| = \text{largest eigenvalue of } A^T A,$$
$$\text{i.e., largest singular value of } A$$

Example 2.3. The Hilbert matrix defined by

$$A = [a_{mn}] = \left[\frac{1}{m+n-1}\right] \tag{E2.3}$$

is notorious for its ill-condition.

We increase the dimension of the Hilbert matrix from $N = 7$ to 12 and make use of the MATLAB commands `cond()` and `det()` to compute the condition number and $\det(A)\det(A^{-1})$ in the MATLAB program "`do_condition`". Especially for $N = 10$, we will see the degree of discrepancy between AA^{-1} and

the identity matrix. Note that the number RCOND following the warning message about near-singularity or ill-condition given by MATLAB is a reciprocal condition number, which can be computed by the rcond() command and is supposed to get close to 1/0 for a well-/badly conditioned matrix.

```
%do_condition.m
clear
for m = 1:6
  for n = 1:6
    A(m,n) = 1/(m+n-1); %A = hilb(6), Eq.(E2.3)
  end
end
for N = 7:12
  for m = 1:N,   A(m,N) = 1/(m + N - 1);  end
  for n = 1:N - 1,   A(N,n) = 1/(N + n - 1);  end
  c = cond(A);  d = det(A)*det(A^- 1);
  fprintf('N = %2d: cond(A) = %e,  det(A)det(A^ - 1) = %8.6f\n', N, c, d);
  if N == 10,  AAI = A*A^ - 1, end
end
```

```
>>do_condition

 N =  7: cond(A) = 4.753674e+008,  det(A)det(A^-1) = 1.000000
 N =  8: cond(A) = 1.525758e+010,  det(A)det(A^-1) = 1.000000
 N =  9: cond(A) = 4.931532e+011,  det(A)det(A^-1) = 1.000001
 N = 10: cond(A) = 1.602534e+013,  det(A)det(A^-1) = 0.999981

 AAI =
 1.0000  0.0000 -0.0001 -0.0000  0.0002 -0.0005  0.0010 -0.0010  0.0004 -0.0001
 0.0000  1.0000 -0.0001 -0.0000  0.0002 -0.0004  0.0007 -0.0007  0.0003 -0.0001
 0.0000  0.0000  1.0000 -0.0000  0.0002 -0.0004  0.0006 -0.0006  0.0003 -0.0000
 0.0000  0.0000 -0.0000  1.0000  0.0001 -0.0003  0.0005 -0.0006  0.0003 -0.0000
 0.0000  0.0000 -0.0000 -0.0000  1.0001 -0.0003  0.0005 -0.0005  0.0002 -0.0000
 0.0000  0.0000 -0.0000 -0.0000  0.0001  0.9998  0.0004 -0.0004  0.0002 -0.0000
 0.0000  0.0000 -0.0000 -0.0000  0.0001 -0.0002  1.0003 -0.0004  0.0002 -0.0000
 0.0000  0.0000 -0.0000 -0.0000  0.0001 -0.0002  0.0003  0.9997  0.0002 -0.0000
 0.0000  0.0000 -0.0000 -0.0000  0.0001 -0.0001  0.0003 -0.0003  1.0001 -0.0000
 0.0000  0.0000 -0.0000 -0.0000  0.0001 -0.0002  0.0003 -0.0003  0.0001  1.0000

 N = 11: cond(A) =5.218389e+014,  det(A)det(A^-1) = 1.000119
         Warning: Matrix is close to singular or badly scaled.
                  Results may be inaccurate. RCOND = 3.659249e-017.
     > In C:\MATLAB\nma\do_condition.m at line 12
 N = 12: cond(A) =1.768065e+016,  det(A)det(A^-1) = 1.015201
```

2.2.3 Gauss–Jordan Elimination

While Gauss elimination consists of forward elimination and backward substitution as explained in Section 2.2.1, Gauss–Jordan elimination consists of forward/backward elimination, which makes the coefficient matrix A an identity matrix so that the resulting RHS vector will appear as the solution.

For simplicity, we start from the triangular matrix–vector equation (2.2.3) obtained by applying the forward elimination:

$$\begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & a_{13}^{(0)} & b_1^{(0)} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & b_2^{(1)} \\ 0 & 0 & a_{33}^{(2)} & b_3^{(2)} \end{bmatrix} \tag{2.2.22}$$

First, we divide the last row by $a_{33}^{(2)}$

$$\begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & a_{13}^{(0)} & b_1^{(0)} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & b_2^{(1)} \\ 0 & 0 & a_{33}^{[1]} = 1 & b_3^{[1]} = b_3^{(2)}/a_{33}^{(2)} \end{bmatrix} \tag{2.2.23}$$

and subtract (the third row $\times a_{m3}^{(m-1)} (m = 1, 2)$) from the above two rows to get

$$\begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & a_{13}^{[1]} = 0 & b_1^{[1]} = b_1^{(0)} - a_{13}^{(0)} b_3^{[1]} \\ 0 & a_{22}^{(1)} & a_{23}^{[1]} = 0 & b_2^{[1]} = b_2^{(1)} - a_{23}^{(1)} b_3^{[1]} \\ 0 & 0 & a_{33}^{[1]} = 1 & b_3^{[1]} \end{bmatrix} \tag{2.2.24}$$

Now, we divide the second row by $a_{22}^{(1)}$:

$$\begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & 0 & b_1^{[1]} \\ 0 & a_{22}^{[2]} = 1 & 0 & b_2^{[2]} = b_2^{[1]}/a_{22}^{[1]} \\ 0 & 0 & a_{33}^{[1]} = 1 & b_3^{[1]} \end{bmatrix} \tag{2.2.25}$$

and subtract (the second row $\times a_{m2}^{(m-1)} (m = 1)$) from the above first row to get

$$\begin{bmatrix} a_{11}^{(0)} & 0 & 0 & b_1^{[2]} = b_1^{[1]} - a_{12}^{(0)} b_2^{[2]} \\ 0 & 1 & 0 & b_2^{[2]} \\ 0 & 0 & 1 & b_3^{[1]} \end{bmatrix} \tag{2.2.26}$$

Lastly, we divide the first row by $a_{11}^{(0)}$ to get

$$\begin{bmatrix} 1 & 0 & 0 & b_1^{[3]} = b_1^{[2]}/a_{11}^{(0)} \\ 0 & 1 & 0 & b_2^{[2]} \\ 0 & 0 & 1 & b_3^{[1]} \end{bmatrix} \tag{2.2.27}$$

which denotes a system of linear equations having an identity matrix as the coefficient matrix

$$I\ \mathbf{x} = \mathbf{b}^{[]} = [\, b_1^{[3]} \quad b_2^{[2]} \quad b_3^{[1]}\,]^T$$

and, consequently, take the RHS vector $\mathbf{b}^{[]}$ as the final solution.

Note that we don't have to distinguish the two steps, the forward/backward elimination. In other words, during the forward elimination, we do the pivoting operations in such a way that the pivot becomes one and other elements above/below the pivot in the same column become zeros.

Consider the following system of linear equations:

$$\begin{bmatrix} -1 & -2 & 2 \\ 1 & 1 & -1 \\ 1 & 2 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix} \tag{2.2.28}$$

We construct the augmented matrix by combining the coefficient matrix and the RHS vector to write

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix} = \begin{bmatrix} -1 & -2 & 2 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & 2 & -1 & 2 \end{bmatrix} \begin{matrix} : & r_1 \\ : & r_2 \\ : & r_3 \end{matrix} \tag{2.2.29}$$

and apply the Gauss–Jordan elimination procedure.

First, we divide the first row r_1 by $a_{11} = -1$ to make the new first row $r_1^{(1)}$ have the pivot $a_{11}^{(1)} = 1$ and subtract $a_{m1} \times r_1^{(1)} (m = 2, 3)$ from the second and third row r_2 and r_3 to get

$$\begin{matrix} r_1 \div (-1) \to \\ r_2 - 1 \times r_1^{(1)} \to \\ r_3 - 1 \times r_1^{(1)} \to \end{matrix} \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & b_1^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} & b_2^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} & b_3^{(1)} \end{bmatrix} = \begin{bmatrix} 1 & 2 & -2 & 1 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{matrix} : & r_1^{(1)} \\ : & r_2^{(1)} \\ : & r_3^{(1)} \end{matrix} \tag{2.2.30a}$$

Then, we divide the second row $r_2^{(1)}$ by $a_{22}^{(1)} = -1$ to make the new second row $r_2^{(2)}$ have the pivot $a_{22}^{(2)} = 1$ and subtract $a_{m2}^{(1)} \times r_2^{(2)} (m = 1, 3)$ from the first and third row $r_1^{(1)}$ and $r_3^{(1)}$ to get

$$\begin{matrix} r_1^{(1)} - 2 \times r_2^{(2)} \to \\ r_2^{(1)} \div (-1) \quad \to \\ r_3^{(1)} - 0 \times r_2^{(2)} \to \end{matrix} \begin{bmatrix} a_{11}^{(2)} & a_{12}^{(2)} & a_{13}^{(2)} & b_1^{(2)} \\ a_{21}^{(2)} & a_{22}^{(2)} & a_{23}^{(2)} & b_2^{(2)} \\ a_{31}^{(2)} & a_{32}^{(2)} & a_{33}^{(2)} & b_3^{(2)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{matrix} : & r_1^{(2)} \\ : & r_2^{(2)} \\ : & r_3^{(2)} \end{matrix} \tag{2.2.30b}$$

Lastly, we divide the third row $r_3^{(2)}$ by $a_{33}^{(2)} = 1$ to make the new third row $r_3^{(3)}$ have the pivot $a_{33}^{(3)} = 1$ and subtract $a_{m3}^{(2)} \times r_3^{(3)} (m = 1, 2)$ from the first and

second row $r_1^{(2)}$ and $r_2^{(2)}$ to get

$$\begin{array}{ll} r_1^{(2)} - 0 \times r_3^{(3)} & \to \\ r_2^{(2)} - (-1) \times r_3^{(3)} & \to \\ r_3^{(2)} & \to \end{array} \begin{bmatrix} a_{11}^{(3)} & a_{12}^{(3)} & a_{13}^{(3)} & b_1^{(3)} \\ a_{21}^{(3)} & a_{22}^{(3)} & a_{23}^{(3)} & b_2^{(3)} \\ a_{31}^{(3)} & a_{32}^{(3)} & a_{33}^{(3)} & b_3^{(3)} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 = x_1 \\ 0 & 1 & 0 & 1 = x_2 \\ 0 & 0 & 1 & 1 = x_3 \end{bmatrix} \begin{array}{l} : r_1^{(3)} \\ : r_2^{(3)} \\ : r_3^{(3)} \end{array} \tag{2.2.30c}$$

After having the identity matrix–vector form like this, we take the RHS vector as the solution.

The general formula applicable for Gauss–Jordan elimination is the same as Eq. (2.2.5), except that the index set is $m \neq k$—that is, all the numbers from $m = 1$ to $m = M$ except $m = k$. Interested readers are recommended to make their own routines to implement this algorithm (see Problem 2.3).

2.3 INVERSE MATRIX

In the previous section, we looked over some algorithms to solve a system of linear equations. We can use such algorithms to solve several systems of linear equations having the same coefficient matrix

$$A\mathbf{x}_1 = \mathbf{b}_1,\ A\mathbf{x}_2 = \mathbf{b}_2, \ldots, A\mathbf{x}_{NB} = \mathbf{b}_{NB}$$

by putting different RHS vectors into one RHS matrix as

$$\begin{gathered} A[\,\mathbf{x}_1 \quad \mathbf{x}_2 \cdots \mathbf{x}_{NB}\,] = [\,\mathbf{b}_1 \quad \mathbf{b}_2 \cdots \mathbf{b}_{NB}\,], \qquad AX = B \\ X = A^{-1}B \end{gathered} \tag{2.3.1}$$

If we substitute an identity matrix I for B into this equation, we will get the matrix inverse $X = A^{-1}I = A^{-1}$. We, however, usually use the MATLAB command `inv(A)` or `A^-1` to compute the inverse of a matrix `A`.

2.4 DECOMPOSITION (FACTORIZATION)

2.4.1 LU Decomposition (Factorization): Triangularization

LU decomposition (factorization) of a nonsingular (square) matrix A means expressing the matrix as the multiplication of a lower triangular matrix L and an upper triangular matrix U, where a lower/upper triangular matrix is a matrix having no nonzero elements above/below the diagonal. For the case where some row switching operation is needed like in the Gauss elimination, we include a permutation matrix P representing the necessary row switching operation(s) to write the LU decomposition as

$$P\ A = L\ U \tag{2.4.1}$$

The usage of a permutation matrix is exemplified by

$$PA = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} a_{31} & a_{32} & a_{33} \\ a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \tag{2.4.2}$$

which denotes switching the first and third rows followed by switching the second and third rows. An interesting and useful property of the permutation matrix is that its transpose agrees with its inverse.

$$P^T P = I, \qquad P^T = P^{-1} \tag{2.4.3}$$

To take a close look at the LU decomposition, we consider a 3×3 nonsingular matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{bmatrix} \tag{2.4.4}$$

First, equating the first rows of both sides yields

$$u_{1n} = a_{1n}, \qquad n = 1, 2, 3 \tag{2.4.5a}$$

Then, equating the second rows of both sides yields

$$a_{21} = l_{21}u_{11}, \quad a_{22} = l_{21}u_{12} + u_{22}, \quad a_{23} = l_{21}u_{13} + u_{23}$$

from which we can get

$$l_{21} = a_{21}/u_{11}, \qquad u_{22} = a_{21} - l_{21}u_{12}, \qquad u_{23} = a_{23} - l_{21}u_{13} \tag{2.4.5b}$$

Now, equating the third rows of both sides yields

$$a_{31} = l_{31}u_{11}, \qquad a_{32} = l_{31}u_{12} + l_{32}u_{22}, \qquad a_{33} = l_{31}u_{13} + u_{32}u_{23} + u_{33}$$

from which we can get

$$l_{31} = a_{31}/u_{11}, \qquad l_{32} = (a_{32} - l_{31}u_{12})/u_{22}, \qquad u_{33} = (a_{33} - l_{31}u_{13}) - l_{32}u_{23} \tag{2.4.5c}$$

In order to put these formulas in one framework to generalize them for matrices having dimension greater than 3, we split this procedure into two steps and write the intermediate lower/upper triangular matrices into one matrix for compactness as

$$\text{step 1: } \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} u_{11} = a_{11} & u_{12} = a_{12} & u_{13} = a_{13} \\ l_{21} = a_{21}/u_{11} & a_{22}^{(1)} = a_{22} - l_{21}u_{12} & a_{23}^{(1)} = a_{23} - l_{21}u_{13} \\ l_{31} = a_{31}/u_{11} & a_{32}^{(1)} = a_{32} - l_{31}u_{12} & a_{33}^{(1)} = a_{33} - l_{31}u_{13} \end{bmatrix} \tag{2.4.6a}$$

$$\text{step 2: } \rightarrow \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} = a_{22}^{(1)} & u_{23} = a_{23}^{(1)} \\ l_{31} & l_{32} = a_{32}^{(1)}/u_{22} & a_{33}^{(2)} = a_{33}^{(1)} - l_{32}u_{23} \end{bmatrix} \tag{2.4.6b}$$

This leads to an LU decomposition algorithm generalized for an $NA \times NA$ nonsingular matrix as described in the following box. The MATLAB routine "`lu_dcmp()`" implements this algorithm to find not only the lower/upper triangular matrix L and U, but also the permutation matrix P. We run it for a 3×3 matrix to get L, U, and P and then reconstruct the matrix $P^{-1}LU = A$ from L, U, and P to ascertain whether the result is right.

```
function [L,U,P] = lu_dcmp(A)
%This gives LU decomposition of A with the permutation matrix P
%  denoting the row switch(exchange) during factorization
NA = size(A,1);
AP = [A  eye(NA)]; %augment with the permutation matrix.
for k = 1:NA - 1
  %Partial Pivoting at AP(k,k)
  [akx, kx] = max(abs(AP(k:NA,k)));
  if akx < eps
     error('Singular matrix and No LU decomposition')
  end
  mx = k+kx-1;
  if kx > 1 % Row change if necessary
    tmp_row = AP(k,:);
    AP(k,:) = AP(mx,:);
    AP(mx,:) = tmp_row;
  end
  % LU decomposition
  for m = k + 1: NA
    AP(m,k) = AP(m,k)/AP(k,k); %Eq.(2.4.8.2)
    AP(m,k+1:NA) = AP(m,k + 1:NA)-AP(m,k)*AP(k,k + 1:NA); %Eq.(2.4.9)
  end
end
P = AP(1:NA, NA + 1:NA + NA); %Permutation matrix
for m = 1:NA
  for n = 1:NA
    if m == n, L(m,m) = 1.;  U(m,m) = AP(m,m);
     elseif m > n, L(m,n) = AP(m,n); U(m,n) = 0.;
     else L(m,n) = 0.; U(m,n) = AP(m,n);
    end
  end
end
if nargout == 0, disp('L*U = P*A with'); L,U,P, end
%You can check if P'*L*U = A?
```

(cf) The number of floating-point multiplications required in this routine `lu_dcmp()` is

$$\sum_{k=1}^{NA-1}(NA-k)(NA-k+1)=\sum_{k=1}^{NA-1}\{NA(NA+1)-(2NA+1)k+k^2\}$$
$$=(NA-1)NA(NA+1)-\frac{1}{2}(2NA+1)(NA-1)NA+\frac{1}{6}(NA-1)NA(2NA-1)$$
$$=\frac{1}{3}(NA-1)NA(NA+1)\approx\frac{1}{3}NA^3 \qquad (2.4.7)$$

with NA: the size of matrix A

0. Initialize $A^{(0)}=A$, or equivalently, $a_{mn}^{(0)}=a_{mn}$ for $m,n=1:NA$.
1. Let $k=1$.
2. If $a_{kk}^{(k-1)}=0$, do an appropriate row switching operation so that $a_{kk}^{(k-1)}\neq 0$.
   When it is not possible, then declare the case of singularity and stop.
3. $a_{kn}^{(k)}=a_{kn}^{(k-1)}=u_{kn}$ for $n=k:NA$ (Just leave the kth row as it is.) (2.4.8a)
   $a_{mk}^{(k)}=a_{mk}^{(k-1)}/a_{kk}^{(k-1)}=l_{mk}$ for $m=k+1:NA$ (2.4.8b)
4. $a_{mn}^{(k)}=a_{mn}^{(k-1)}-a_{mk}^{(k)}a_{kn}^{(k)}$ for $m,n=k+1:NA$ (2.4.9)
5. Increment k by 1 and if $k<NA-1$, go to step 1; otherwise, go to step 6.
6. Set the part of the matrix $A^{(NA-1)}$ below the diagonal to L (lower triangular matrix with the diagonal of 1's) and the part on and above the diagonal to U (upper triangular matrix).

```
>>A = [1 2 5;0.2 1.6 7.4; 0.5 4 8.5];
>>[L,U,P] = lu_dcmp(A) %LU decomposition
  L = 1.0  0    0    U =  1   2   5    P = 1   0   0
      0.5  1.0  0         0   3   6        0   0   1
      0.2  0.4  1.0       0   0   4        0   1   0
>>P'*L*U - A %check the validity of the result (P' = P^-1)
  ans =  0     0     0
         0     0     0
         0     0     0
>>[L,U,P] = lu(A) %for comparison with the MATLAB built-in function
```

What is the LU decomposition for? It can be used for solving a system of linear equations as

$$A\mathbf{x}=\mathbf{b} \qquad (2.4.10)$$

Once we have the LU decomposition of the coefficient matrix $A=P^TLU$, it is more efficient to use the lower/upper triangular matrices for solving Eq. (2.4.10)

than to apply the Gauss elimination method. The procedure is as follows:

$$P^T LU\ \mathbf{x} = \mathbf{b}, \qquad LU\ \mathbf{x} = P\ \mathbf{b}, \quad U\ \mathbf{x} = L^{-1}\ P\ \mathbf{b}, \quad \mathbf{x} = U^{-1}\ L^{-1}\ P\ \mathbf{b} \tag{2.4.11}$$

Note that the premultiplication of L^{-1} and U^{-1} by a vector can be performed by the forward and backward substitution, respectively. The following program "`do_lu_dcmp.m`" applies the LU decomposition method, the Gauss elimination algorithm, and the MATLAB operators '\' and '`inv`' or '`^-1`' to solve Eq. (2.4.10), where A is the five-dimensional Hilbert matrix (introduced in Example 2.3) and $\mathbf{b} = A\mathbf{x}^o$ with $\mathbf{x}^o = [\,1 \quad 1 \quad 1 \quad 1 \quad 1\,]^T$. The residual error $\|A\mathbf{x}_i - \mathbf{b}\|$ of the solutions obtained by the four methods and the numbers of floating-point operations required for carrying out them are listed in Table 2.1. The table shows that, once the inverse matrix A^{-1} is available, the inverse matrix method requiring only N^2 multiplications/additions (N is the dimension of the coefficient matrix or the number of unknown variables) is the most efficient in computation, but the worst in accuracy. Therefore, if we need to continually solve the system of linear equations with the same coefficient matrix A for different RHS vectors, it is a reasonable choice in terms of computation time and accuracy to save the LU decomposition of the coefficient matrix A and apply the forward/backward substitution process.

```
%do_lu_dcmp
% Use LU decomposition, Gauss elimination to solve Ax = b
A = hilb(5);
[L,U,P] = lu_dcmp(A); %LU decomposition
x = [1 -2 3 -4 5 -6 7 -8 9 -10]';
b = A*x(1:size(A,1));
flops(0), x_lu = backsubst(U,forsubst(L,P*b)); %Eq.(2.4.11)
flps(1) = flops; % assuming that we have already got L\U decomposition
flops(0), x_gs = gauss(A,b); flps(3) = flops;
flops(0), x_bs = A\b; flps(4) = flops;
AI = A^-1; flops(0), x_iv = AI*b; flps(5) = flops;
% assuming that we have already got the inverse matrix
disp('     x_lu         x_gs        x_bs        x_iv')
format short e
solutions = [x_lu x_gs x_bs x_iv]
errs = [norm(A*x_lu - b) norm(A*x_gs - b) norm(A*x_bs - b) norm(A*x_iv - b)]
format short, flps
```

```
function x = forsubst(L,B)
%forward substitution for a lower-triangular matrix equation Lx = B
N = size(L,1);
x(1,:) = B(1,:)/L(1,1);
for m = 2:N
   x(m,:) = (B(m,:)-L(m,1:m - 1)*x(1:m-1,:))/L(m,m);
end
```

```
function x = backsubst(U,B)
%backward substitution for a upper-triangular matrix equation Ux = B
N = size(U,2);
x(N,:) = B(N,:)/U(N,N);
for m = N-1: -1:1
   x(m,:) = (B(m,:) - U(m,m + 1:N)*x(m + 1:N,:))/U(m,m);
end
```

Table 2.1 Residual Error and the Number of Floating-Point Operations of Various Solutions

	`tmp = forsubst(L,P*b)` `backsubst(U,tmp)`	`gauss(A,b)`	`A\b`	`A^-1*b`
$\|\|A\mathbf{x}_i - \mathbf{b}\|\|$	1.3597e-016	5.5511e-017	1.7554e-016	3.0935e-012
# of flops	123	224	155	50

(cf) The numbers of flops for the LU decomposition and the inverse of the matrix A are not counted.

(cf) Note that the command 'flops' to count the number of floating-point operations is no longer available in MATLAB 6.x and higher versions.

2.4.2 Other Decomposition (Factorization): Cholesky, QR, and SVD

There are several other matrix decompositions such as Cholesky decomposition, QR decomposition, and singular value decomposition (SVD). Instead of looking into the details of these algorithms, we will simply survey the MATLAB built-in functions implementing these decompositions.

Cholesky decomposition factors a positive definite symmetric/Hermitian matrix into an upper triangular matrix premultiplied by its transpose as

$$A = U^T U \qquad (U\text{: an upper triangular matrix}) \tag{2.4.12}$$

and is implemented by the MATLAB built-in function `chol()`.

(cf) If a (complex-valued) matrix A satisfies $A^{*T} = A$—that is, the conjugate transpose of a matrix equals itself—it is said to be Hermitian. It is said to be just symmetric in the case of a real-valued matrix with $A^T = A$.

(cf) If a square matrix A satisfies $\mathbf{x}^{*T} A\ \mathbf{x} > 0\ \forall\ \mathbf{x} \neq \mathbf{0}$, the matrix is said to be positive definite (see Appendix B).

```
>>A = [2 3 4;3 5 6;4 6 9]; %a positive definite symmetric matrix
>>U = chol(A) %Cholesky decomposition
   U = 1.4142    2.1213    2.8284
       0         0.7071    0.0000
       0         0         1.0000
>>U'*U - A %to check if the result is right
```

QR decomposition is to express a square or rectangular matrix as the product of an orthogonal (unitary) matrix Q and an upper triangular matrix R as

$$A = QR \tag{2.4.13}$$

where $Q^T Q = I$ ($Q^{*T} Q = I$). This is implemented by the MATLAB built-in function `qr()`.

(cf) If all the columns of a (complex-valued) matrix A are orthonormal to each other—that is, $A^{*T}A = I$, or, equivalently, $A^{*T} = A^{-1}$—it is said to be unitary. It is said to be orthogonal in the case of real-valued matrix with $A^T = A^{-1}$.

SVD (singular value decomposition) is to express an $M \times N$ matrix A in the following form

$$A = USV^T \tag{2.4.14}$$

where U is an orthogonal (unitary) $M \times M$ matrix, V is an orthogonal (unitary) $N \times N$ matrix, and S is a real diagonal $M \times N$ matrix having the singular values of A (the square roots of the eigenvalues of $A^T A$) in decreasing order on its diagonal. This is implemented by the MATLAB built-in function `svd()`.

```
>>A = [1 2;2 3;3 5]; %a rectangular matrix
>>[U,S,V] = svd(A) %Singular Value Decomposition
  U = 0.3092  0.7557 -0.5774   S = 7.2071  0        V = 0.5184  -0.8552
      0.4998 -0.6456 -0.5774       0       0.2403       0.8552   0.5184
      0.8090  0.1100  0.5774       0       0
>>err = U*S*V'-A %to check if the result is right
  err = 1.0e-015* -0.2220  -0.2220
                        0        0
                   0.4441        0
```

2.5 ITERATIVE METHODS TO SOLVE EQUATIONS

2.5.1 Jacobi Iteration

Let us consider the equation

$$3x + 1 = 0$$

which can be cast into an iterative scheme as

$$2x = -x - 1; x = -\frac{x+1}{2} \rightarrow x_{k+1} = -\frac{1}{2}x_k - \frac{1}{2}$$

Starting from some initial value x_0 for $k = 0$, we can incrementally change k by 1 each time to proceed as follows:

$$\begin{aligned}
x_1 &= -2^{-1} - 2^{-1}x_0 \\
x_2 &= -2^{-1} - 2^{-1}x_1 = -2^{-1} + 2^{-2} + 2^{-2}x_0 \\
x_3 &= -2^{-1} - 2^{-1}x_2 = -2^{-1} + 2^{-2} - 2^{-3} - 2^{-3}x_0 \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots
\end{aligned}$$

Whatever the initial value x_0 is, this process will converge to the sum of a geometric series with the ratio of $(-1/2)$ as

$$x_k = \frac{a_0}{1-r} = \frac{-1/2}{1-(-1/2)} = -\frac{1}{3} = x^0 \qquad \text{as } k \to \infty$$

and what is better, the limit is the very true solution to the given equation. We are happy with this, but might feel uneasy, because we are afraid that this convergence to the true solution is just a coincidence. Will it always converge, no matter how we modify the equation so that only x remains on the LHS?

To answer this question, let us try another iterative scheme.

$$x = -2x - 1 \to x_{k+1} = -2x_k - 1$$
$$x_1 = -1 - 2x_0$$
$$x_2 = -1 - 2x_1 = -1 - 2(-1 - 2x_0) = -1 + 2 + 2^2 x_0$$
$$x_3 = -1 - 2x_2 = -1 + 2 - 2^2 - 2^3 x_0$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

This iteration will diverge regardless of the initial value x_0. But, we are never disappointed, since we know that no one can be always lucky.

To understand the essential difference between these two cases, we should know the fixed-point theorem (Section 4.1). Apart from this, let's go into a system of equations.

$$\begin{bmatrix} 3 & 2 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \qquad A\mathbf{x} = \mathbf{b}$$

Dividing the first equation by 3 and transposing all term(s) other than x_1 to the RHS and dividing the second equation by 2 and transposing all term(s) other than x_2 to the RHS, we have

$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \end{bmatrix} = \begin{bmatrix} 0 & -2/3 \\ -1/2 & 0 \end{bmatrix} \begin{bmatrix} x_{1,k} \\ x_{2,k} \end{bmatrix} + \begin{bmatrix} 1/3 \\ -1/2 \end{bmatrix}$$
$$\mathbf{x}_{k+1} = \widetilde{A}\,\mathbf{x}_k + \widetilde{\mathbf{b}} \tag{2.5.1}$$

Assuming that this scheme works well, we set the initial value to zero ($\mathbf{x}_0 = \mathbf{0}$) and proceed as

$$\mathbf{x}_k \to [I + \widetilde{A} + \widetilde{A}^2 + \cdots]\widetilde{\mathbf{b}} = [I - A]^{-1}\widetilde{\mathbf{b}} = \begin{bmatrix} 1 & 2/3 \\ 1/2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1/3 \\ -1/2 \end{bmatrix}$$
$$= \frac{1}{1-1/3} \begin{bmatrix} 1 & -2/3 \\ -1/2 & 1 \end{bmatrix} \begin{bmatrix} 1/3 \\ -1/2 \end{bmatrix} = \frac{1}{2/3} \begin{bmatrix} 2/3 \\ -2/3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \mathbf{x}^o \tag{2.5.2}$$

which will converge to the true solution $\mathbf{x}^o = [1 \;\; -1]^T$. This suggests another method of solving a system of equations, which is called Jacobi iteration. It can be generalized for an $N \times N$ matrix–vector equation as follows:

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mm}x_m + \cdots + a_{mN}x_N = b_m$$

$$x_m^{(k+1)} = -\sum_{n\neq m}^{N} \frac{a_{mn}}{a_{mm}} x_n^{(k)} + \frac{b_m}{a_{mm}} \qquad \text{for } m = 1, 2, \ldots, N$$

$$\mathbf{x}_{k+1} = \widetilde{A}\ \mathbf{x}_k + \widetilde{\mathbf{b}} \quad \text{for each time stage } k \tag{2.5.3}$$

where

$$\widetilde{A}_{N\times N} = \begin{bmatrix} 0 & -a_{12}/a_{11} & \cdots & -a_{1N}/a_{11} \\ -a_{21}/a_{22} & 0 & \cdots & -a_{2N}/a_{22} \\ \cdot & \cdot & \cdots & \cdot \\ -a_{N1}/a_{NN} & -a_{N2}/a_{NN} & \cdots & 0 \end{bmatrix}, \ \widetilde{\mathbf{b}} = \begin{bmatrix} b_1/a_{11} \\ b_2/a_{22} \\ \cdot \\ b_N/a_{NN} \end{bmatrix}$$

This scheme is implemented by the following MATLAB routine "`jacobi()`". We run it to solve the above equation.

```
function  X = jacobi(A,B,X0,kmax)
%This function finds a soltuion to Ax = B by Jacobi iteration.
if nargin < 4, tol = 1e-6; kmax = 100; %called by jacobi(A,B,X0)
 elseif kmax < 1, tol = max(kmax,1e-16); kmax = 100; %jacobi(A,B,X0,tol)
 else tol = 1e-6;  %jacobi(A,B,X0,kmax)
end
if nargin < 3, X0 = zeros(size(B)); end
NA = size(A,1);
X = X0;  At = zeros(NA,NA);
for m = 1:NA
   for n = 1:NA
      if n ~= m, At(m,n) = -A(m,n)/A(m,m); end
   end
   Bt(m,:) = B(m,:)/A(m,m);
end
for k = 1: kmax
   X = At*X + Bt; %Eq. (2.5.3)
   if nargout == 0, X, end %To see the intermediate results
   if norm(X - X0)/(norm(X0) + eps) < tol, break; end
   X0 = X;
end
```

```
>>A = [3 2;1 2]; b = [1  -1]'; %the coefficient matrix and RHS vector
>>x0 = [0  0]'; %the initial value
>>x = jacobi(A,b,x0,20) %to repeat 20 iterations starting from x0
   x = 1.0000
      -1.0000
>>jacobi(A,b,x0,20) %omit output argument to see intermediate results
   X = 0.3333    0.6667    0.7778    0.8889    0.9259  ......
      -0.5000   -0.6667   -0.8333   -0.8889   -0.9444  ......
```

2.5.2 Gauss–Seidel Iteration

Let us take a close look at Eq. (2.5.1). Each iteration of Jacobi method updates the whole set of N variables at a time. However, so long as we do not use a

multiprocessor computer capable of parallel processing, each one of N variables is updated sequentially one by one. Therefore, it is no wonder that we could speed up the convergence by using all the most recent values of variables for updating each variable even in the same iteration as follows:

$$x_{1,k+1} = -\frac{2}{3}x_{2,k} + \frac{1}{3}$$
$$x_{2,k+1} = -\frac{1}{2}x_{1,k+1} - \frac{1}{2}$$

This scheme is called Gauss–Seidel iteration, which can be generalized for an $N \times N$ matrix–vector equation as follows:

$$x_m^{(k+1)} = \frac{b_m - \sum_{n=1}^{m-1} a_{mn}x_n^{(k+1)} - \sum_{n=m+1}^{N} a_{mn}x_n^{(k)}}{a_{mm}}$$
$$\text{for } m = 1, \ldots, N \text{ and for each time stage } k \tag{2.5.4}$$

This is implemented in the following MATLAB routine "`gauseid()`", which we will use to solve the above equation.

```
function  X = gauseid(A,B,X0,kmax)
%This function finds x = A^-1 B by Gauss–Seidel iteration.
if nargin < 4, tol = 1e-6; kmax = 100;
  elseif kmax < 1, tol = max(kmax,1e-16); kmax = 1000;
  else tol = 1e-6;
end if nargin < 4, tol = 1e-6; kmax = 100; end
if nargin < 3, X0 = zeros(size(B)); end
NA = size(A,1); X = X0;
for k = 1: kmax
   X(1,:) = (B(1,:)-A(1,2:NA)*X(2:NA,:))/A(1,1);
   for m = 2:NA-1
      tmp = B(m,:)-A(m,1:m-1)*X(1:m - 1,:)-A(m,m + 1:NA)*X(m + 1:NA,:);
      X(m,:) = tmp/A(m,m); %Eq.(2.5.4)
   end
   X(NA,:) = (B(NA,:)-A(NA,1:NA - 1)*X(1:NA - 1,:))/A(NA,NA);
   if nargout == 0, X, end %To see the intermediate results
   if norm(X - X0)/(norm(X0) + eps)<tol, break; end
   X0 = X;
end
```

```
>>A = [3 2;1 2]; b = [1  -1]'; %the coefficient matrix and RHS vector
>>x0 = [0  0]'; %the initial value
>>gauseid(A,b,x0,10) %omit output argument to see intermediate results
   X = 0.3333    0.7778    0.9259    0.9753    0.9918  ......
      -0.6667   -0.8889   -0.9630   -0.9877   -0.9959  ......
```

As with the Jacobi iteration in the previous section, we can see this Gauss–Seidel iteration converging to the true solution $\mathbf{x}^o = [1 \;\; -1]^T$ and that with fewer iterations. But, if we use a multiprocessor computer capable of parallel processing,

the Jacobi iteration may be better in speed even with more iterations, since it can exploit the advantage of simultaneous parallel computation.

Note that the Jacobi/Gauss–Seidel iterative scheme seems unattractive and even unreasonable if we are given a standard form of linear equations as

$$A\mathbf{x} = \mathbf{b}$$

because the computational overhead for converting it into the form of Eq. (2.5.3) may be excessive. But, it is not always the case, especially when the equations are given in the form of Eq. (2.5.3)/(2.5.4). In such a case, we simply repeat the iterations without having to use such ready-made routines as "`jacobi()`" or "`gauseid()`". Let us see the following example.

Example 2.4. Jacobi or Gauss–Seidel Iterative Scheme. Suppose the temperature of a metal rod of length 10 m has been measured to be 0°C and 10°C at each end, respectively. Find the temperatures x_1, x_2, x_3, and x_4 at the four points equally spaced with the interval of 2 m, assuming that the temperature at each point is the average of the temperatures of both neighboring points.

We can formulate this problem into a system of equations as

$$x_1 = \frac{x_0 + x_2}{2}, \quad x_2 = \frac{x_1 + x_3}{2}, \quad x_3 = \frac{x_2 + x_4}{2},$$
$$x_4 = \frac{x_3 + x_5}{2} \qquad \text{with } x_0 = 0 \text{ and } x_5 = 10 \tag{E2.4}$$

This can easily be cast into Eq. (2.5.3) or Eq. (2.5.4) as programmed in the following program "`nm2e04.m`":

```
%nm2e04
N = 4; %the number of unknown variables/equations
kmax = 20; tol = 1e-6;
At = [0 1 0 0; 1 0 1 0; 0 1 0 1; 0 0 1 0]/2;
x0 = 0; x5 = 10; %boundary values
b = [x0/2 0 0 x5/2]'; %RHS vector
%initialize all the values to the average of boundary values
xp=ones(N,1)*(x0 + x5)/2;
%Jacobi iteration
for k = 1:kmax
   x = At*xp +b; %Eq.(E2.4)
   if norm(x - xp)/(norm(xp)+eps) < tol, break; end
   xp = x;
end
k, xj = x
%Gauss–Seidel iteration
xp = ones(N,1)*(x0 + x5)/2; x = xp; %initial value
for k = 1:kmax
   for n = 1:N, x(n) = At(n,:)*x + b(n); end %Eq.(E2.4)
   if norm(x - xp)/(norm(xp) + eps) < tol, break; end
   xp = x;
end
k, xg = x
```

The following example illustrates that the Jacobi iteration and the Gauss–Seidel iteration can also be used for solving a system of nonlinear equations, although there is no guarantee that it will work for every nonlinear equation.

Example 2.5. Gauss–Seidel Iteration for Solving a Set of Nonlinear Equations.
We are going to use the Gauss–Seidel iteration to solve a system of nonlinear equations as

$$\begin{aligned} x_1^2 + 10x_1 + 2x_2^2 - 13 &= 0 \\ 2x_1^3 - x_2^2 + 5x_2 - 6 &= 0 \end{aligned} \tag{E2.5.1}$$

In order to do so, we convert these equations into the following form, which suits the Gauss–Seidel scheme.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} (13 - x_1^2 - 2x_2^2)/10 \\ (6 - 2x_1^3 + x_2^2)/5 \end{bmatrix} \tag{E2.5.2}$$

We make the MATLAB program "nm2e05.m", which uses the Gauss–Seidel iteration to solve these equations. Interested readers are recommended to run this program to see that this simple iteration yields the solution within the given tolerance of error in just six steps. How marvelous it is to solve the system of nonlinear equations without any special algorithm!

(cf) Due to its remarkable capability to deal with a system of nonlinear equations, the Gauss–Seidel iterative method plays an important role in solving partial differential equations (see Chapter 9).

```
%nm2e05.m
% use Gauss-Seidel iteration to solve a set of nonlinear equations
clear
kmax = 100; tol = 1e-6;
x = zeros(2,1); %initial value
for k = 1:kmax
   xp = x; % to remember the previous solution
   x(1) = (13 - x(1)^2 - 2*x(2)^2)/10; % (E2.5.2)
   x(2) = (6 - x(1)^3)/5;
   if norm(x - xp)/(norm(xp) + eps)<tol, break; end
end
k, x
```

2.5.3 The Convergence of Jacobi and Gauss–Seidel Iterations

Jacobi and Gauss–Seidel iterations have a very simple computational structure because they do not need any matrix inversion. So, it may be of practical use, if only the convergence is guaranteed. However, everything cannot always be fine,

as illustrated in Section 2.5.1. Then, what is the convergence condition? It is the diagonal dominancy of coefficient matrix A, which is stated as follows:

$$|a_{mm}| > \sum_{n \neq m}^{N} |a_{mn}| \qquad \text{for } m = 1, 2, \ldots, N \tag{2.5.5}$$

This implies that the convergence of the iterative schemes is ensured if, in each row of coefficient matrix A, the absolute value of the diagonal element is greater than the sum of the absolute values of the other elements. It should be noted, however, that this is a sufficient, not a necessary, condition. In other words, the iterative scheme may work even if the above condition is not strictly satisfied.

One thing to note is the relaxation technique, which may be helpful in accelerating the convergence of Gauss–Seidel iteration. It is a slight modification of Eq. (2.5.4) as

$$x_m^{(k+1)} = (1-\omega)x_m^{(k)} + \omega \frac{b_m - \sum_{n=1}^{m-1} a_{mn} x_n^{(k+1)} - \sum_{n=m+1}^{N} a_{mn} x_n^{(k)}}{a_{mm}}$$
$$\text{with } 0 < \omega < 2 \tag{2.5.6}$$

and is called SOR (successive overrelaxation) for the relaxation factor $1 < \omega < 2$ and successive underrelaxation for $0 < \omega < 1$. But regrettably, there is no general rule for selecting the optimal value of the relaxation factor ω.

PROBLEMS

2.1 Recursive Least-Squares Estimation (RLSE)

(a) Run the program '`do_rlse.m`' (in Section 2.1.4) with another value of the true parameter

```
xo = [1 2]'
```

What is the parameter estimate obtained from the RLS solution?

(b) Run the program "`do_rlse`" with a small matrix P like

```
P = 0.01*eye(NA);
```

What is the parameter estimate obtained from the RLS solution? Is it still close to the value of the true parameter?

(c) Insert the statements in the following box at appropriate places in the MATLAB code "`do_rlse.m`" appeared in Section 2.1.4. Remove the last two statements and run it to compare the times required for using the RLS solution and the standard LS solution to get the parameter estimates on-line.

```
%nm2p01.m
.. .. .. .. .. .. .. ..
time_on = 0; time_off = 0;
.. .. .. .. .. .. .. ..
tic
.. .. .. .. .. .. .. ..
time_on = time_on + toc;
tic
xk_off = A\b; %standard LS solution
time_off = time_off + toc;
.. .. .. .. .. .. .. ..
solutions = [x xk_off]
discrepancy = norm(x - xk_off)
times = [time_on time_off]
```

2.2 Delicacy of Scaled Partial Pivoting

As a complement to Example 2.2, we want to compare no pivoting, partial pivoting, scaled partial pivoting, and full pivoting in order to taste the delicacy of row switching strategy. To do it in a systematic way, add the third input argument (`pivoting`) to the Gauss elimination routine '`gauss()`' and modify its contents by inserting the following statements into appropriate places so that the new routine "`gauss(A,b,pivoting)`" implements the partial pivoting procedure optionally depending on the value of '`pivoting`'. You can also remove any unnecessary parts.

```
- if nargin < 3, pivoting = 2; end %scaled partial pivoting by default
- switch pivoting
   case 2, [akx,kx] = max(abs(AB(k:NA,k))./...
             max(abs([AB(k:NA,k + 1:NA) eps*ones(NA - k + 1,1)]'))');
   otherwise, [akx,kx] = max(abs(AB(k:NA,k)));  %partial pivoting
  end
- &pivoting > 0 %partial pivoting not to be done for pivot = 1
```

(a) Use this routine with `pivoting = 0/1/2`, the '\' operator and the '`inv()`' command to solve the systems of linear equations with the coefficient matrices and the RHS vectors shown below and fill in Table P2.2 with the residual error $||A_i\mathbf{x} - \mathbf{b}_i||$ to compare the results in terms of how well the solutions satisfy the equation, that is, $||A_i\mathbf{x} - \mathbf{b}_i|| \approx 0$.

(1) $A_1 = \begin{bmatrix} 10^{-15} & 1 \\ 1 & 10^{11} \end{bmatrix}, \qquad \mathbf{b}_1 = \begin{bmatrix} 1 + 10^{-15} \\ 10^{11} + 1 \end{bmatrix}$

(2) $A_2 = \begin{bmatrix} 10^{-14.6} & 1 \\ 1 & 10^{15} \end{bmatrix}, \qquad \mathbf{b}_2 = \begin{bmatrix} 1 + 10^{-14.6} \\ 10^{15} + 1 \end{bmatrix}$

(3) $A_3 = \begin{bmatrix} 10^{11} & 1 \\ 1 & 10^{-15} \end{bmatrix}, \qquad \mathbf{b}_3 = \begin{bmatrix} 10^{11} + 1 \\ 1 + 10^{-15} \end{bmatrix}$

Table P2.2 Comparison of gauss() with Different Pivoting Methods in Terms of $||A\mathbf{x}_i - \mathbf{b}||$

	$A_1\mathbf{x} = \mathbf{b}_1$	$A_2\mathbf{x} = \mathbf{b}_2$	$A_3\mathbf{x} = \mathbf{b}_3$	$A_4\mathbf{x} = \mathbf{b}_4$
`gauss(A,b,0)` (no pivoting)	1.25e-01			
`gauss(A,b,1)` (partial pivoting)		4.44e-16		
`gauss(A,b,2)` (scaled partial pivoting)			0	
`A\b`				6.25e-02
`A^-1*b`				

(4) $A_4 = \begin{bmatrix} 10^{14.6} & 1 \\ 1 & 10^{-15} \end{bmatrix}, \qquad \mathbf{b}_4 = \begin{bmatrix} 10^{14.6} + 1 \\ 1 + 10^{-15} \end{bmatrix}$

(b) Which pivoting strategy yields the worst result for problem (1) in (a)? Has the row swapping been done during the process of partial pivoting and scaled partial pivoting? If yes, did it work to our advantage? Did the '\' operator or the '`inv()`' command give you any better result?

(c) Which pivoting strategy yields the worst result for problem (2) in (a)? Has the row swapping been done during the process of partial pivoting and scaled partial pivoting? If yes, did it produce a positive effect for this case? Did the '\' operator or the '`inv()`' command give you any better result?

(d) Which pivoting strategy yields the best result for problem (3) in (a)? Has the row swapping been done during the process of partial pivoting and scaled partial pivoting? If yes, did it produce a positive effect for this case?

(e) The coefficient matrix A_3 is the same as would be obtained by applying the full pivoting scheme for A_1 to have the largest pivot element. Does the full pivoting give better result than no pivoting or the (scaled) partial pivoting?

(f) Which pivoting strategy yields the best result for problem (4) in (a)? Has the row swapping been done during the process of partial pivoting and scaled partial pivoting? If yes, did it produce a positive effect for this case? Did the '\' operator or the '`inv()`' command give you any better result?

2.3 Gauss–Jordan Elimination Algorithm Versus Gauss Elimination Algorithm

Gauss–Jordan elimination algorithm mentioned in Section 2.2.3 is trimming the coefficient matrix A into an identity matrix and then takes the RHS vector/matrix as the solution, while Gauss elimination algorithm introduced with the corresponding routine "`gauss()`" in Section 2.2.1 makes the matrix an upper-triangular one and performs backward substitution to get the solution. Since Gauss–Jordan elimination algorithm does not need backward substitution, it seems to be simpler than Gauss elimination algorithm.

Table P2.3 Comparison of Several Methods for Solving a Set of Linear Equations

	`gauss(A,b)`	`gaussj(A,b)`	`A\b`	`A^-1*b`
$\|\|A\mathbf{x}_i - \mathbf{b}\|\|$	3.1402e-016		8.7419e-016	
# of flops	1124	1744	785	7670

(a) Modify the routine "`gauss()`" into a routine "`gaussj()`" which implements Gauss–Jordan elimination algorithm and count the number of multiplications consumed by the routine, excluding those required for partial pivoting. Compare it with the number of multiplications consumed by "`gauss()`" [Eq. (2.2.18)]. Does it support or betray our expectation that Gauss–Jordan elimination would take fewer computations than Gauss elimination?

(b) Use both of the routines, the '\' operator and the '`inv()`' command or '`^-1`' to solve the system of linear equations

$$A\mathbf{x} = \mathbf{b} \tag{P2.3.1}$$

where A is the 10-dimensional Hilbert matrix (see Example 2.3) and $\mathbf{b} = A\mathbf{x}^o$ with $\mathbf{x}^o = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]^T$. Fill in Table P2.3 with the residual errors

$$||A\mathbf{x}_i - \mathbf{b}|| \approx 0 \tag{P2.3.2}$$

as a way of describing how well each solution satisfies the equation.

(cf) The numbers of floating-point operations required for carrying out the computations are listed in Table P2.3 so that readers can compare the computational loads of different approaches. Those data were obtained by using the MATLAB command `flops()`, which is available only in MATLAB of version below 6.0.

2.4 Tridiagonal System of Linear Equations

Consider the following system of linear equations:

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 && = b_1\\
&a_{21}x_1 + a_{22}x_2 + a_{23}x_3 && = b_2\\
&\qquad\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots && \\
&a_{N-1,N-2}x_{N-2} + a_{N-1,N-1}x_{N-1} + a_{N-1,N}x_N && = b_{N-1}\\
&a_{N,N-1}x_{N-1} + a_{N,N}x_N && = b_N
\end{aligned} \tag{P2.4.1}$$

which can be written in a compact form by using a matrix–vector notation as

$$A_{N\times N}\mathbf{x} = \mathbf{b} \tag{P2.4.2}$$

Table P2.4 The Computational Load of the Methods to Solve a Tri-diagonal System of Equations

	`gauss(A,b)`	`trid(A,b)`	`gauseid()`	`gauseid1()`	`A\b`
# of flops	141	50	2615	2082	94

where

$$A_{N\times N} = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 0 & a_{N-1,N-2} & a_{N-1,N-1} & a_{N-1,N} \\ 0 & 0 & 0 & a_{N,N-1} & a_{NN} \end{bmatrix},$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ x_{N-1} \\ x_N \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ b_{N-1} \\ b_N \end{bmatrix}$$

This is called a tridiagonal system of equations on account of that the coefficient matrix A has nonzero elements only on its main diagonal and super-/subdiagonals.

(a) Modify the Gauss elimination routine "`gauss()`" (Section 2.2.1) in such a way that this special structure can be exploited for reducing the computational burden. Give the name '`trid()`' to the modified routine and save it in an m-file named "`trid.m`" for future use.

(b) Modify the Gauss–Seidel iteration routine "`gauseid()`" (Section 2.5.2) in such a way that this special structure can be exploited for reducing the computational burden. Let the name of the modified routine be "`Gauseid1()`".

(c) Noting that Eq. (E2.4) in Example 2.4 can be trimmed into a tridiagonal structure as (P2.4.2), use the routines "`gauss()`", "`trid()`", "`gauseid()`", "`gauseid1()`", and the backslash (\) operator to solve the problem.

(cf) The numbers of floating-point operations required for carrying out the computations are listed in Table P2.4 so that readers can compare the computational loads of the different approaches.

2.5 LU Decomposition of a Tridiagonal Matrix

Modify the LU decomposition routine "`lu_dcmp()`" (Section 2.4.1) in such a way that the tridiagonal structure can be exploited for reducing the

computational burden. Give the name "`lu_trid()`" to the modified routine and use it to get the LU decomposition of the tridiagonal matrix

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \tag{P2.5.1}$$

You may type the following statements into the MATLAB command window:

```
>>A = [2 -1 0 0; -1 2 -1 0; 0 -1 2 -1; 0 0 -1 2];
>>[L,U] = lu_trid(A)
>>L*U - A % = 0 (No error)?
```

2.6 LS Solution by Backslash Operator and QR Decomposition

The backslash ('`A\b`') operator and the matrix left division ('`mldivide(A,b)`') function turn out to be the most efficient means for solving a system of linear equations as Eq. (P2.3.1). They are also capable of dealing with the under/over-determined cases. Let's see how they handle the under/over-determined cases.

(a) For an underdetermined system of linear equations

$$A_1\mathbf{x} = \mathbf{b}_1, \qquad \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 14 \\ 32 \end{bmatrix} \tag{P2.6.1}$$

find the minimum-norm solution (2.1.7) and the solutions that can be obtained by typing the following statements in the MATLAB command window:

```
>>A1 = [1 2 3; 4 5 6]; b1 = [14 32]';
>>x_mn = A1'*(A1*A1')^-1*b1, x_pi = pinv(A1)*b1, x_bs = A1\b1
```

Are the three solutions the same?

(b) For another underdetermined system of linear equations

$$A_2\mathbf{x} = \mathbf{b}_2, \qquad \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 14 \\ 28 \end{bmatrix} \tag{P2.6.2}$$

find the solutions by using Eq. (2.1.7), the commands `pinv()`, and backslash (\). If you are not pleased with the result obtained from Eq. (2.1.7), you can remove one of the two rows from the coefficient matrix A_2 and try again. Identify the minimum solution(s). Are the equations redundant or inconsistent?

Table P2.6.1 Comparison of Several Methods for Computing the LS Solution

	QR	LS: Eq. (2.1.10)	`pinv(A)*b`	`A\b`
$\|\|A\mathbf{x}_i - \mathbf{b}\|\|$	2.8788e-016		2.8788e-016	
# of flops	25	89	196	92

(c) For another underdetermined system of linear equations

$$A_2\mathbf{x} = \mathbf{b}_3, \qquad \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 21 \\ 21 \end{bmatrix} \tag{P2.6.3}$$

find the solutions by using Eq. (2.1.7), the commands `pinv()`, and backslash (\). Does any of them satisfy Eq. (P2.6.3) closely? Are the equations redundant or inconsistent?

(d) For an overdetermined system of linear equations

$$A_4\mathbf{x} = \mathbf{b}_4, \qquad \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5.2 \\ 7.8 \\ 2.2 \end{bmatrix} \tag{P2.6.4}$$

find the LS (least-squares) solution (2.1.10), that can be obtained from the following statements. Fill in the corresponding blanks of Table P2.6.1 with the results.

```
>>A4 = [1 2; 2 3; 4 -1]; b4 = [5.2 7.8 2.2]';
>> x_ls = (A4'*A4)\A4'*b4, x_pi = pinv(A4)*b4, x_bs = A4\b4
```

(e) We can use QR decomposition to solve a system of linear equations as Eq. (P2.3.1), where the coefficient matrix A is square and nonsingular or rectangular with the row dimension greater than the column dimension. The procedure is explained as follows:

$$A\mathbf{x} = QR\mathbf{x} = \mathbf{b}, \qquad R\mathbf{x} = Q^{-1}\mathbf{b} = Q'\mathbf{b}, \qquad \mathbf{x} = R^{-1}Q'\mathbf{b} \tag{P2.6.5}$$

Note that $Q'Q = I$; $Q' = Q^{-1}$ (orthogonality) and the premultiplication of R^{-1} can be performed by backward substitution, because R is an upper-triangular matrix. You are supposed not to count the number of floating-point operations needed for obtaining the LU and QR decompositions, assuming that they are available.

(i) Apply the QR decomposition, the LU decomposition, Gauss elimination, and the backslash (\) operator to solve the system of linear

Table P2.6.2 Comparison of Several Methods for Solving a System of Linear Equations

	LU	QR	`gauss(A,b)`	`A\b`
$\|\|A\mathbf{x}_i - \mathbf{b}\|\|$		7.8505e-016		8.7419e-016
# of flops	453	327	1124	785

equations whose coefficient matrix is the 10-dimensional Hilbert matrix (see Example 2.3) and fill in the corresponding blanks of Table P2.6.2 with the results.

(ii) Apply the QR decomposition to solve the system of linear equations given by Eq. (P2.6.4) and fill in the corresponding blanks of Table P2.6.2 with the results.

(cf) This problem illustrates that QR decomposition is quite useful for solving a system of linear equations, where the coefficient matrix A is square and nonsingular or rectangular with the row dimension greater than the column dimension and no rank deficiency.

2.7 Cholesky Factorization of a Symmetric Positive Definite Matrix:

If a matrix A is symmetric and positive definite, we can find its LU decomposition such that the upper triangular matrix U is the transpose of the lower triangular matrix L, which is called Cholesky factorization.

Consider the Cholesky factorization procedure for a 4×4 matrix

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{12} & a_{22} & a_{23} & a_{24} \\ a_{13} & a_{23} & a_{33} & a_{34} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix} = \begin{bmatrix} u_{11} & 0 & 0 & 0 \\ u_{12} & u_{22} & 0 & 0 \\ u_{13} & u_{23} & u_{33} & 0 \\ u_{14} & u_{24} & u_{34} & u_{44} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

$$= \begin{bmatrix} u_{11}^2 & u_{11}u_{12} & u_{11}u_{13} & u_{11}u_{14} \\ u_{12}u_{11} & u_{12}^2 + u_{22}^2 & u_{12}u_{13} + u_{22}u_{23} & u_{12}u_{14} + u_{22}u_{24} \\ u_{13}u_{11} & u_{13}u_{12} + u_{23}u_{22} & u_{13}^2 + u_{23}^2 + u_{33}^2 & u_{13}u_{14} + u_{23}u_{24} + u_{33}u_{34} \\ u_{14}u_{11} & u_{14}u_{12} + u_{24}u_{22} & u_{14}u_{13} + u_{24}u_{23} + u_{34}u_{33} & u_{14}^2 + u_{24}^2 + u_{34}^2 + u_{44}^2 \end{bmatrix} \tag{P2.7.1}$$

Equating every row of the matrices on both sides yields

$$u_{11} = \sqrt{a_{11}}, \qquad u_{12} = a_{12}/u_{11},\ u_{13} = a_{13}/u_{11},\ u_{14} = a_{14}/u_{11} \tag{P2.7.2.1}$$

$$u_{22} = \sqrt{a_{22} - u_{12}^2}, \qquad u_{23} = (a_{23} - u_{13}u_{12})/u_{22},\ u_{24} = (a_{24} - u_{14}u_{12})/u_{22} \tag{P2.7.2.2}$$

$$u_{33} = \sqrt{a_{33} - u_{23}^2 - u_{13}^2}, \qquad u_{34} = (a_{43} - u_{24}u_{23} - u_{14}u_{13})/u_{33} \tag{P2.7.2.3}$$

$$u_{44} = \sqrt{a_{44} - u_{34}^2 - u_{24}^2 - u_{14}^2} \tag{P2.7.2.4}$$

which can be combined into two formulas as

$$u_{kk} = \sqrt{a_{kk} - \sum_{i=1}^{k-1} u_{ik}^2} \quad \text{for } k = 1 : N \tag{P2.7.3a}$$

$$u_{km} = \left(a_{km} - \sum_{i=1}^{k-1} u_{im}u_{ik}\right) / u_{kk} \quad \text{for } m = k+1 : N \text{ and } k = 1 : N \tag{P2.7.3b}$$

(a) Make a MATLAB routine "`cholesky()`", which implements these formulas to perform Cholesky factorization.

(b) Try your routine "`cholesky()`" for the following matrix and check if $U^T U - A \approx \mathrm{O}$ (U: the upper triangular matrix). Compare the result with that obtained by using the MATLAB built-in routine "`chol()`".

$$A = \begin{bmatrix} 1 & 2 & 4 & 7 \\ 2 & 13 & 23 & 38 \\ 4 & 23 & 77 & 122 \\ 7 & 38 & 122 & 294 \end{bmatrix} \tag{P2.7.4}$$

(c) Use the routine "`lu_dcmp()`" and the MATLAB built-in routine "`lu()`" to get the LU decomposition for the above matrix (P2.7.4) and check if $P^T LU - A \approx \mathrm{O}$, where L and U are the lower/upper triangular matrix, respectively. Compare the result with that obtained by using the MATLAB built-in routine "`lu()`".

2.8 Usage of SVD (Singular Value Decomposition)

What is SVD good for? Suppose we have the singular value decomposition of an $M \times N$ real-valued matrix A as

$$A = USV^T \tag{P2.8.1}$$

where U is an orthogonal $M \times M$ matrix, V an orthogonal $N \times N$ matrix, and S a real diagonal $M \times N$ matrix having the singular value σ_i's of A (the square roots of the eigenvalues of $A^T A$) in decreasing order on its diagonal. Then, it is possible to improvise the pseudo-inverse even in the case of rank-deficient matrices (with rank(A) < min(M, N)) for which the left/right pseudo-inverse can't be found. The virtual pseudo-inverse can be written as

$$\hat{A}^{-1} = \hat{V}\hat{S}^{-1}\hat{U}^T \tag{P2.8.2}$$

where $\hat{S}^{-1}$ is the diagonal matrix having $1/\sigma_i$ on its diagonal that is reconstructed by removing all-zero(-like) rows/columns of the matrix S and substituting $1/\sigma_i$ for $\sigma_i \neq 0$ into the resulting matrix; $\hat{V}$ and $\hat{U}$ are reconstructed by removing the columns of V and U corresponding to the zero singular value(s). Consequently, SVD has a specialty in dealing with the singular cases. Let us take a closer look at this through the following problems.

(a) Consider the problem of solving

$$A_1\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 12 \end{bmatrix} = \mathbf{b}_1 \tag{P2.8.3}$$

Since this belongs to the underdetermined case ($M = 2 < 3 = N$), it seems that we can use Eq. (2.1.7) to find the minimum-norm solution.

(i) Type the following statements into the MATLAB command window.

```
>>A1 = [1 2 3; 2 4 6]; b1 = [6;12]; x = A1'*(A1*A1')^-1*b1 %Eq. (2.1.7)
```

What is the result? Explain why it is so and support your answer by typing

```
>>r = rank(A1)
```

(ii) Type the following statements into the MATLAB command window to see the SVD-based minimum-norm solution. What is the value of $\mathbf{x} = \hat{A}_1^{-1}\mathbf{b}_1 = \hat{V}\hat{S}^{-1}\hat{U}^T\mathbf{b}_1$ and $\|A_1\mathbf{x} - \mathbf{b}_1\|$?

```
[U,S,V] = svd(A1); %(P2.8.1)
u = U(:,1:r); v = V(:,1:r); s = S(1:r,1:r);
AIp = v*diag(1./diag(s))*u'; %faked pseudo-inverse (P2.8.2)
x = AIp*b1 %minimum-norm solution for singular underdetermined
err = norm(A1*x - b1) %residual error
```

(iii) To see that the norm of this solution is less than that of any other solution which can be obtained by adding any vector in the null space of the coefficient matrix A_1, type the following statements into the MATLAB command window. What is implied by the result?

```
nullA = null(A1); normx = norm(x);
for n = 1:1000
  if norm(x + nullA*(rand(size(nullA,2),1)-0.5)) < normx
    disp('What the hell smaller-norm sol - not minimum norm');
  end
end
```

(b) For the problem

$$A_2\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 9 \end{bmatrix} = \mathbf{b}_2 \tag{P2.8.4}$$

compare the minimum-norm solution based on SVD and that obtained by Eq. (2.1.7).

(c) Consider the problem of solving

$$A_3\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 9 \\ 7 & 11 & 18 \\ -2 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \mathbf{b}_3 \qquad \text{(P2.8.5)}$$

Since this belongs to the overdetermined case ($M = 4 > 3 = N$), it seems that we can use Eq. (2.1.10) to find the LS (least-squares) solution.

(i) Type the following statements into the MATLAB command window:

```
>>A3=[1 2 3; 4 5 9;7 11 18;-2 3 1];
>>b3=[1;2;3;4]; x=(A3'*A3)^-1*A3'*b3 %Eq. (2.1.10)
```

What is the result? Explain why it is so in connection with the rank of A_3.

(ii) Similarly to (a)(ii), find the SVD-based least-squares solution.

```
[U,S,V] = svd(A3);
u=U(:,1:r); v = V(:,1:r); s = S(1:r,1:r);
AIp = v*diag(1./diag(s))*u'; x = AIp*b
```

(iii) To see that the residual error of this solution is less than that of any other vector around it, type the following statements into the MATLAB command window. What is implied by the result?

```
err = norm(A3*x-b3)
for n = 1:1000
   if norm(A3*(x+rand(size(x))-0.5)-b)<err
     disp('What the hell smaller error sol - not LSE?');
   end
end
```

(d) For the problem

$$A_4\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 9 \\ 7 & 11 & -1 \\ -2 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \mathbf{b}_4 \qquad \text{(P2.8.6)}$$

compare the LS solution based on SVD and that obtained by Eq. (2.1.10).

(cf) This problem illustrates that SVD can be used for fabricating a universal solution of a set of linear equations, minimum-norm or least-squares, for all the possible rank deficiency of the coefficient matrix A.

2.9 Gauss–Seidel Iterative Method with Relaxation Technique

(a) Try the relaxation technique (introduced in Section 2.5.3) with several values of the relaxation factor $\omega = 0.2, 0.4, \ldots, 1.8$ for the following problems. Find the best one among these values of the relaxation factor for each problem, together with the number of iterations required for satisfying the termination criterion $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|/\|\mathbf{x}_k\| < 10^{-6}$.

(i) $$A_1\mathbf{x} = \begin{bmatrix} 5 & -4 \\ -9 & 10 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \mathbf{b}_1 \tag{P2.9.1}$$

(ii) $$A_2\mathbf{x} = \begin{bmatrix} 2 & -1 \\ -1 & 4 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \mathbf{b}_2 \tag{P2.9.2}$$

(iii) The nonlinear equations (E2.5.1) given in Example 2.5.

(b) Which of the two matrices A_1 and A_2 has stronger diagonal dominancy in the above equations? For which equation does Gauss–Seidel iteration converge faster, Eq. (P2.9.1) or Eq. (P2.9.2)? What would you conjecture about the relationship between the convergence speed of Gauss–Seidel iteration for a set of linear equations and the diagonal dominancy of the coefficient matrix A?

(c) Is the relaxation technique always helpful for improving the convergence speed of the Gauss–Seidel iterative method regardless of the value of the relaxation factor ω?

3

INTERPOLATION AND CURVE FITTING

There are two topics to be dealt with in this chapter, namely, interpolation[1] and curve fitting. Interpolation is to connect discrete data points in a plausible way so that one can get reasonable estimates of data points between the given points. The interpolation curve goes through all data points. Curve fitting, on the other hand, is to find a curve that could best indicate the trend of a given set of data. The curve does not have to go through the data points. In some cases, the data may have different accuracy/reliability/uncertainty and we need the weighted least-squares curve fitting to process such data.

3.1 INTERPOLATION BY LAGRANGE POLYNOMIAL

For a given set of $N+1$ data points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_N, y_N)\}$, we want to find the coefficients of an Nth-degree polynomial function to match them:

$$p_N(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_N x^N \tag{3.1.1}$$

The coefficients can be obtained by solving the following system of linear equations.

$$\begin{aligned} a_0 + x_0 a_1 + x_0^2 a_2 + \cdots + x_0^N a_N &= y_0 \\ a_0 + x_1 a_1 + x_1^2 a_2 + \cdots + x_1^N a_N &= y_1 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots &\quad . \\ a_0 + x_N a_1 + x_N^2 a_2 + \cdots + x_N^N a_N &= y_N \end{aligned} \tag{3.1.2}$$

[1] If we estimate the values of the unknown function at the points that are inside/outside the range of collected data points, we call it the interpolation/extrapolation.

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

But, as the number of data points increases, so does the number of unknown variables and equations, consequently, it may be not so easy to solve. That is why we look for alternatives to get the coefficients $\{a_0, a_1, \ldots, a_N\}$.

One of the alternatives is to make use of the Lagrange polynomials

$$l_N(x) = y_0 \frac{(x - x_1)(x - x_2)\cdots(x - x_N)}{(x_0 - x_1)(x_0 - x_2)\cdots(x_0 - x_N)} + y_1 \frac{(x - x_0)(x - x_2)\cdots(x - x_N)}{(x_1 - x_0)(x_1 - x_2)\cdots(x_1 - x_N)} + \cdots + y_N \frac{(x - x_0)(x - x_1)\cdots(x - x_{N-1})}{(x_N - x_0)(x_N - x_1)\cdots(x_N - x_{N-1})}$$

$$l_N(x) = \sum_{m=0}^{N} y_m L_{N,m}(x) \text{ with } L_{N,m}(x) = \frac{\prod_{k\neq m}^{N}(x - x_k)}{\prod_{k\neq m}^{N}(x_m - x_k)} = \prod_{k\neq m}^{N} \frac{x - x_k}{x_m - x_k} \quad (3.1.3)$$

It can easily be shown that the graph of this function matches every data point

$$l_N(x_m) = y_m \qquad \forall\ m = 0, 1, \ldots, N \quad (3.1.4)$$

since the Lagrange coefficient polynomial $L_{N,m}(x)$ is 1 only for $x = x_m$ and zero for all other data points $x = x_k$ $(k \neq m)$. Note that the Nth-degree polynomial function matching the given $N + 1$ points is unique and so Eq. (3.1.1) having the coefficients obtained from Eq. (3.1.2) must be the same as the Lagrange polynomial (3.1.3).

Now, we have the MATLAB routine "`lagranp()`" which finds us the coefficients of Lagrange polynomial (3.1.3) together with each Lagrange coefficient polynomial $L_{N,m}(x)$. In order to understand this routine, you should know that MATLAB deals with polynomials as their coefficient vectors arranged in descending order and the multiplication of two polynomials corresponds to the convolution of the coefficient vectors as mentioned in Section 1.1.6.

```
function [l,L] = lagranp(x,y)
%Input : x = [x0 x1 ... xN], y = [y0 y1 ... yN]
%Output: l = Lagrange polynomial coefficients of degree N
%        L = Lagrange coefficient polynomial
N = length(x)-1; %the degree of polynomial
l = 0;
for m = 1:N + 1
  P = 1;
  for k = 1:N + 1
    if k ~= m, P = conv(P,[1 -x(k)])/(x(m)-x(k)); end
  end
  L(m,:) = P; %Lagrange coefficient polynomial
  l = l + y(m)*P; %Lagrange polynomial (3.1.3)
end
```

```
%do_lagranp.m
x = [-2 -1 1 2]; y = [-6 0 0 6]; % given data points
l = lagranp(x,y) % find the Lagrange polynomial
xx = [-2: 0.02 : 2]; yy = polyval(l,xx); %interpolate for [-2,2]
clf, plot(xx,yy,'b', x,y,'*') %plot the graph
```

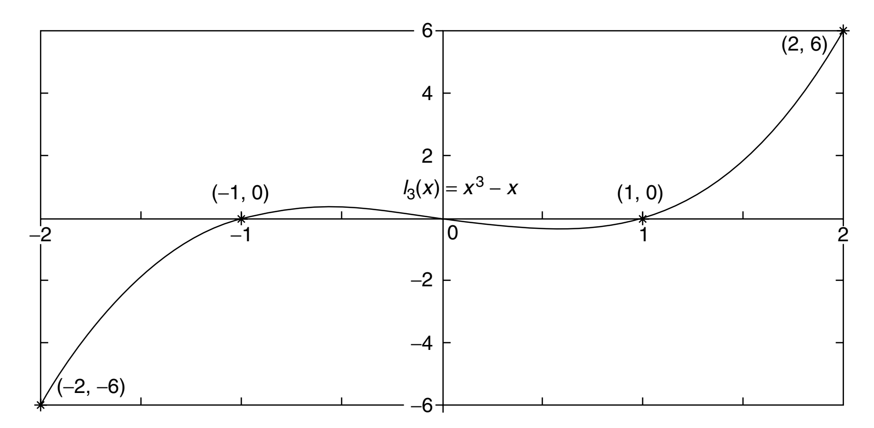


Figure 3.1 The graph of a third-degree Lagrange polynomial.

We make the MATLAB program "`do_lagranp.m`" to use the routine "`lagranp()`" for finding the third-degree polynomial $l_3(x)$ which matches the four given points

$$\{(-2, -6), (-1, 0), (1, 0), (2, 6)\}$$

and to check if the graph of $l_3(x)$ really passes the four points. The results from running this program are depicted in Fig. 3.1.

```
>>do_lagranp
  l =  1    0   -1    0 % meaning l3(x) = 1 · x^3 + 0 · x^2 − 1 · x + 0
```

3.2 INTERPOLATION BY NEWTON POLYNOMIAL

Although the Lagrange polynomial works pretty well for interpolation irrespective of the interval widths between the data points along the x-axis, it requires restarting the whole computation with heavier burden as data points are appended. Differently from this, the Nth-degree Newton polynomial matching the $N + 1$ data points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_N, y_N)\}$ can be recursively obtained as the sum of the $(N - 1)$th-degree Newton polynomial matching the N data points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{N-1}, y_{N-1})\}$ and one additional term.

$$\begin{aligned} n_N(x) &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots \\ &= n_{N-1}(x) + a_N(x - x_0)(x - x_1)\cdots(x - x_{N-1}) \qquad \text{with } n_0(x) = a_0 \end{aligned} \tag{3.2.1}$$

In order to derive a formula to find the successive coefficients $\{a_0, a_1, \ldots, a_N\}$ that make this equation accommodate the data points, we will determine a_0 and a_1 so that

$$n_1(x) = n_0(x) + a_1(x - x_0) \tag{3.2.2}$$

matches the first two data points (x_0, y_0) and (x_1, y_1). We need to solve the two equations

$$n_1(x_0) = a_0 + a_1(x_0 - x_0) = y_0$$
$$n_1(x_1) = a_0 + a_1(x_1 - x_0) = y_1$$

to get

$$a_0 = y_0, \qquad a_1 = \frac{y_1 - a_0}{x_1 - x_0} = \frac{y_1 - y_0}{x_1 - x_0} \equiv Df_0 \tag{3.2.3}$$

Starting from this first-degree Newton polynomial, we can proceed to the second-degree Newton polynomial

$$n_2(x) = n_1(x) + a_2(x - x_0)(x - x_1) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) \tag{3.2.4}$$

which, with the same coefficients a_0 and a_1 as (3.2.3), still matches the first two data points (x_0, y_0) and (x_1, y_1), since the additional (third) term is zero at (x_0, y_0) and (x_1, y_1). This is to say that the additional polynomial term does not disturb the matching of previous existing data. Therefore, given the additional matching condition for the third data point (x_2, y_2), we only have to solve

$$n_2(x_2) = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) \equiv y_2$$

for only one more coefficient a_2 to get

$$a_2 = \frac{y_2 - a_0 - a_1(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)} = \frac{y_2 - y_0 - \dfrac{y_1 - y_0}{x_1 - x_0}(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)}$$
$$= \frac{y_2 - y_1 + y_1 - y_0 - \dfrac{y_1 - y_0}{x_1 - x_0}(x_2 - x_1 + x_1 - x_0)}{(x_2 - x_0)(x_2 - x_1)}$$
$$= \frac{\dfrac{y_2 - y_1}{x_2 - x_1} - \dfrac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0} = \frac{Df_1 - Df_0}{x_2 - x_0} \equiv D^2 f_0 \tag{3.2.5}$$

Generalizing these results (3.2.3) and (3.2.5) yields the formula to get the Nth coefficient a_N of the Newton polynomial function (3.2.1) as

$$a_N = \frac{D^{N-1} f_1 - D^{N-1} f_0}{x_N - x_0} \equiv D^N f_0 \tag{3.2.6}$$

This is the divided difference, which can be obtained successively from the second row of Table 3.1.

Table 3.1 Divided Difference Table

x_k	y_k	Df_k	$D^2 f_k$	$D^3 f_k$	—
x_0	y_0	$Df_0 = \frac{y_1 - y_0}{x_1 - x_0}$	$D^2 f_0 = \frac{Df_1 - Df_0}{x_2 - x_0}$	$D^3 f_0 = \frac{D^2 f_1 - D^2 f_0}{x_3 - x_0}$	—
x_1	y_1	$Df_1 = \frac{y_2 - y_1}{x_2 - x_1}$	$D^2 f_1 = \frac{Df_2 - Df_1}{x_3 - x_1}$	—	
x_2	y_2	$Df_2 = \frac{y_3 - y_2}{x_3 - x_2}$	—		
x_3	y_3	—			

```
function [n,DD] = newtonp(x,y)
%Input : x = [x0 x1 ... xN]
%        y = [y0 y1 ... yN]
%Output: n = Newton polynomial coefficients of degree N
N = length(x)-1;
DD = zeros(N + 1,N + 1);
DD(1:N + 1,1) = y';
for k = 2:N + 1
  for m = 1: N + 2 - k %Divided Difference Table
    DD(m,k) = (DD(m + 1,k - 1) - DD(m,k - 1))/(x(m + k - 1)- x(m));
  end
end
a = DD(1,:); %Eq.(3.2.6)
n = a(N+1); %Begin with Eq.(3.2.7)
for k = N:-1:1 %Eq.(3.2.7)
  n = [n a(k)] - [0 n*x(k)]; %n(x)*(x - x(k - 1))+a_k - 1
end
```

Note that, as mentioned in Section 1.3, it is of better computational efficiency to write the Newton polynomial (3.2.1) in the nested multiplication form as

$$n_N(x) = ((\cdots(a_N(x - x_{N-1}) + a_{N-1})(x - x_{N-2}) + \cdots) + a_1)(x - x_0) + a_0 \tag{3.2.7}$$

and that the multiplication of two polynomials corresponds to the convolution of the coefficient vectors as mentioned in Section 1.1.6. We make the MATLAB routine "`newtonp()`" to compose the divided difference table like Table 3.1 and construct the Newton polynomial for a set of data points.

For example, suppose we are to find a Newton polynomial matching the following data points

$$\{(-2, -6), (-1, 0), (1, 0), (2, 6), (4, 60)\}$$

From these data points, we construct the divided difference table as Table 3.2 and then use this table together with Eq. (3.2.1) to get the Newton polynomial

Table 3.2 Divided differences

x_k	y_k	Df_k	$D^2 f_k$	$D^3 f_k$	$D^4 f_k$
−2	−6	$\frac{0-(-6)}{-1-(-2)}=6$	$\frac{0-6}{1-(-2)}=-2$	$\frac{2-(-2)}{2-(-2)}=1$	$\frac{1-1}{4-(-2)}=0$
−1	0	$\frac{0-0}{1-(-1)}=0$	$\frac{6-0}{2-(-1)}=2$	$\frac{7-2}{4-(-1)}=1$	
1	0	$\frac{6-0}{2-1}=6$	$\frac{27-6}{4-1}=7$		
2	6	$\frac{60-6}{4-2}=27$			
4	60				

as follows:

$$\begin{aligned}
n(x) &= y_0 + Df_0(x-x_0) + D^2 f_0(x-x_0)(x-x_1) \\
&\quad + D^3 f_0(x-x_0)(x-x_1)(x-x_2) + 0 \\
&= -6 + 6(x-(-2)) - 2(x-(-2))(x-(-1)) \\
&\quad + 1(x-(-2))(x-(-1))(x-1) \\
&= -6 + 6(x+2) - 2(x+2)(x+1) + (x+2)(x^2-1) \\
&= x^3 + (-2+2)x^2 + (6-6-1)x - 6 + 12 - 4 - 2 = x^3 - x
\end{aligned}$$

We might begin with not necessarily the first data point, but, say, the third one (1,0), and proceed as follows to end up with the same result.

$$\begin{aligned}
n(x) &= y_2 + Df_2(x-x_2) + D^2 f_2(x-x_2)(x-x_3) \\
&\quad + D^3 f_2(x-x_2)(x-x_3)(x-x_4) + 0 \\
&= 0 + 6(x-1) + 7(x-1)(x-2) + 1(x-1)(x-2)(x-4) \\
&= 6(x-1) + 7(x^2-3x+2) + (x^2-3x+2)(x-4) \\
&= x^3 + (7-7)x^2 + (6-21+14)x - 6 + 14 - 8 = x^3 - x
\end{aligned}$$

This process is cast into the MATLAB program "`do_newtonp.m`", which illustrates that the Newton polynomial (3.2.1) does not depend on the order of the data points; that is, changing the order of the data points does not make any difference.

```
%do_newtonp.m
x = [-2 -1  1  2  4]; y = [-6  0 0 6 60];
n = newtonp(x,y) %l = lagranp(x,y) for comparison
x = [-1 -2  1  2  4]; y = [ 0 -6 0 6 60];
n1 = newtonp(x,y) %with the order of data changed for comparison
xx = [-2:0.02: 2]; yy = polyval(n,xx);
clf, plot(xx,yy,'b-',x,y,'*')
```

Now, let us see the interpolation problem from the viewpoint of approximation. For this purpose, suppose we are to approximate some function, say,

$$f(x) = \frac{1}{1 + 8x^2}$$

by a polynomial. We first pick up some sample points on the graph of this function, such as listed below, and look for the polynomial functions $n_4(x)$, $n_8(x)$, and $n_{10}(x)$ to match each of the three sets of points, respectively.

x_k	−1.0	−0.5	0	0.5	1.0
y_k	1/9	1/3	1	1/3	1/9

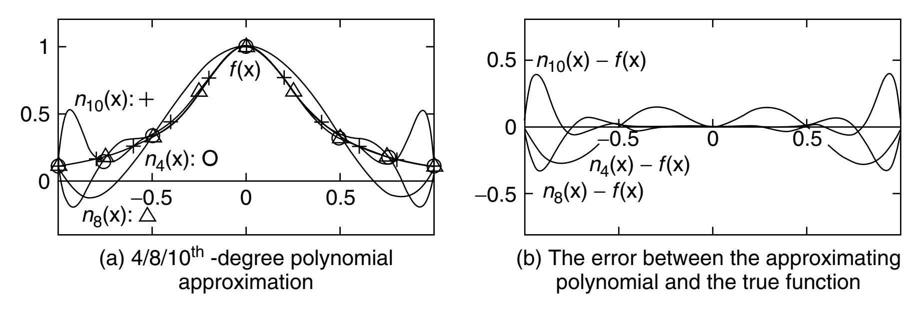
x_k	−1.0	−0.75	−0.5	−0.25	0	0.25	0.5	0.75	1.0
y_k	1/9	2/11	1/3	2/3	1	2/3	1/3	2/11	1/9

x_k	−1.0	−0.8	−0.6	−0.4	−0.2	0	0.2	0.4	0.6	0.8	1.0
y_k	1/9	25/153	25/97	25/57	25/33	1	25/33	25/57	25/97	25/153	1/9

We made the MATLAB program "`do_newtonp1.m`" to do this job and plot the graphs of the polynomial functions together with the graph of the true function $f(x)$ and their error functions separately for comparison as depicted in Fig. 3.2, where the parts for $n_8(x)$ and $n_{10}(x)$ are omitted to provide the readers with some room for practice.

```
%do_newtonp1.m –  plot Fig.3.2
x = [-1 -0.5  0  0.5  1.0]; y = f31(x);
n = newtonp(x,y)
xx = [-1:0.02: 1]; %the interval to look over
yy = f31(xx); %graph of the true function
yy1 = polyval(n,xx); %graph of the approximate polynomial function
subplot(221), plot(xx,yy,'k-', x,y,'o', xx,yy1,'b')
subplot(222), plot(xx,yy1-yy,'r') %graph of the error function
```

```
function y = f31(x)
y=1./(1+8*x.^2);
```



(a) 4/8/10th -degree polynomial approximation

(b) The error between the approximating polynomial and the true function

Figure 3.2 Interpolation from the viewpoint of approximation.

Remark 3.1. Polynomial Wiggle and Runge Phenomenon. Here is one thing to note. Strangely, increasing the degree of polynomial contributes little to reducing the approximation error. Rather contrary to our usual expectation, it tends to make the oscillation strikingly large, which is called the polynomial wiggle and the error gets bigger in the parts close to both ends as can be seen in Fig. 3.2, which is called the Runge phenomenon. That is why polynomials of degree 5 or above are seldom used for the purpose of interpolation, unless they are sure to fit the data.

3.3 APPROXIMATION BY CHEBYSHEV POLYNOMIAL

At the end of the previous section, we considered a polynomial approximation problem of finding a polynomial close to a given (true) function $f(x)$ and have the freedom to pick up the target points $\{x_0, x_1, \ldots, x_N\}$ in our own way. Once the target points have been fixed, it is nothing but an interpolation problem that can be solved by the Lagrange or Newton polynomial.

In this section, we will think about how to choose the target points for better approximation, rather than taking equidistant points along the x axis. Noting that the error tends to get bigger in the parts close to both ends of the interval when we chose the equidistant target points, it may be helpful to set the target points denser in the parts close to both ends than in the middle part. In this context, a possible choice is the projection (onto the x axis) of the equidistant points on the circle centered at the middle point of the interval along the x axis (see Fig. 3.3). That is, we can choose in the normalized interval $[-1, +1]$

$$x_k' = \cos\frac{2N+1-2k}{2(N+1)}\pi \qquad \text{for } k = 0, 1, \ldots, N \tag{3.3.1a}$$

and for an arbitrary interval $[a, b]$,

$$\mathrm{x_k} = \frac{\mathrm{b-a}}{2}\mathrm{x_k'} + \frac{\mathrm{a+b}}{2} = \frac{\mathrm{b-a}}{2}\cos\frac{\mathrm{2N+1-2k}}{\mathrm{2(N+1)}}\pi + \frac{\mathrm{a+b}}{2} \qquad \text{for k} = 0, 1, \ldots, \mathrm{N} \tag{3.3.1b}$$

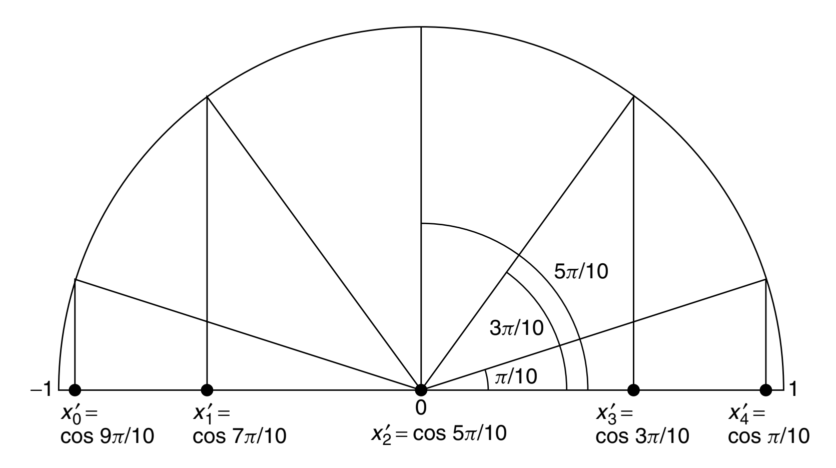


Figure 3.3 Chebyshev nodes ($N = 4$).

which are referred to as the Chebyshev nodes. The approximating polynomial obtained on the basis of these Chebyshev nodes is called the Chebyshev polynomial.

Let us try the Chebyshev nodes on approximating the function

$$f(x) = \frac{1}{1 + 8x^2}$$

We can set the 5/9/11 Chebyshev nodes by Eq. (3.3.1) and get the Lagrange or Newton polynomials $c_4(x)$, $c_8(x)$, and $c_{10}(x)$ matching these target points, which are called the Chebyshev polynomial. We make the MATLAB program "`do_lagnewch.m`" to do this job and plot the graphs of the polynomial functions together with the graph of the true function $f(x)$ and their error functions separately for comparison as depicted in Fig. 3.4. The parts for $c_8(x)$ and $c_{10}(x)$ are omitted to give the readers a chance to practice what they have learned in this section.

```
%do_lagnewch.m –  plot Fig.3.4
N = 4; k = [0:N];
x=cos((2*N + 1 - 2*k)*pi/2/(N + 1)); %Chebyshev nodes(Eq.(3.3.1))
y=f31(x);
c=newtonp(x,y) %Chebyshev polynomial
xx = [-1:0.02: 1]; %the interval to look over
yy = f31(xx); %graph of the true function
yy1 = polyval(c,xx); %graph of the approximate polynomial function
subplot(221), plot(xx,yy,'k-', x,y,'o', xx,yy1,'b')
subplot(222), plot(xx,yy1-yy,'r') %graph of the error function
```

Comparing Fig. 3.4 with Fig. 3.2, we see that the maximum deviation of the Chebyshev polynomial from the true function is considerably less than that of

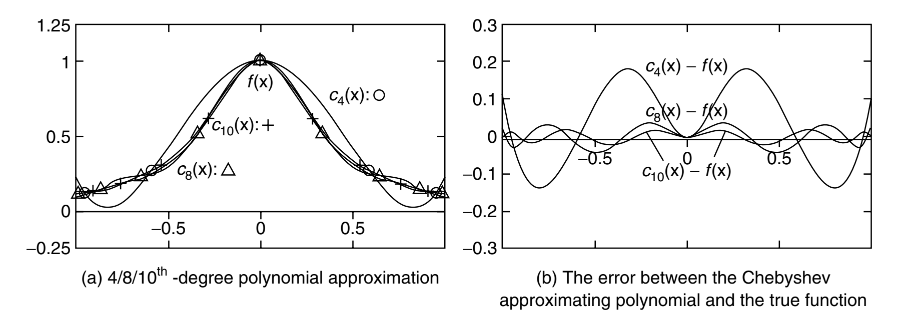


Figure 3.4 Approximation using the Chebyshev polynomial.

Lagrange/Newton polynomial with equidistant nodes. It can also be seen that increasing the number of the Chebyshev nodes—or, equivalently, increasing the degree of Chebyshev polynomial—makes a substantial contribution towards reducing the approximation error.

There are several things to note about the Chebyshev polynomial.

Remark 3.2. Chebyshev Nodes and Chebyshev Coefficient Polynomials $T_m(x)$

1. The Chebyshev coefficient polynomial is defined as

$$T_{N+1}(x') = \cos((N+1)\cos^{-1} x') \qquad \text{for} -1 \le x' \le +1 \tag{3.3.2}$$

and the Chebyshev nodes defined by Eq. (3.3.1a) are actually zeros of this function:

$$T_{N+1}(x') = \cos((N+1)\cos^{-1} x') = 0, \quad (N+1)\cos^{-1} x' = (2k'+1)\pi/2$$

2. Equation (3.3.2) can be written via the trigonometric formula in a recursive form as

$$\begin{aligned} T_{N+1}(x') &= \cos(\cos^{-1} x' + N\cos^{-1} x') \\ &= \cos(\cos^{-1} x')\cos(N\cos^{-1} x') - \sin(\cos^{-1} x')\sin(N\cos^{-1} x') \\ &= x' T_N(x') + \frac{1}{2}\{\cos((N+1)\cos^{-1} x') - \cos((N-1)\cos^{-1} x')\} \\ &= x' T_N(x') + \frac{1}{2}T_{N+1}(x') - \frac{1}{2}T_{N-1}(x') \end{aligned}$$

$$T_{N+1}(x') = 2x' T_N(x') - T_{N-1}(x') \quad \text{for} \quad N \ge 1 \tag{3.3.3a}$$

$$T_0(x') = \cos 0 = 1, \qquad T_1(x') = \cos(\cos^{-1} x') = x' \tag{3.3.3b}$$

3. At the Chebyshev nodes x'_k defined by Eq. (3.3.1a), the set of Chebyshev coefficient polynomials

$$\{T_0(x'), T_1(x'), \ldots, T_N(x')\}$$

are orthogonal in the sense that

$$\sum_{k=0}^{N} T_m(x_k^{'})T_n(x_k^{'}) = 0 \qquad \text{for } m \neq n \tag{3.3.4a}$$

$$\sum_{k=0}^{N} T_m^2(x_k^{'}) = \frac{N+1}{2} \qquad \text{for } m \neq 0 \tag{3.3.4b}$$

$$\sum_{k=0}^{N} T_0^2(x_k^{'}) = N+1 \qquad \text{for } m = 0 \tag{3.3.4c}$$

4. The Chebyshev coefficient polynomials $T_{N+1}(x^{'})$ for up to $N = 6$ are collected in Table 3.3, and their graphs are depicted in Fig. 3.5. As can be seen from the table or the graph, the Chebyshev coefficient polynomials of even/odd degree $(N+1)$ are even/odd functions and have an equi-ripple characteristic with the range of $[-1, +1]$, and the number of rising/falling (intervals) within the domain of $[-1, +1]$ is $N+1$.

We can make use of the orthogonality [Eq. (3.3.4)] of Chebyshev coefficient polynomials to derive the Chebyshev polynomial approximation formula.

$$f(x) \cong c_N(x) = \sum_{m=0}^{N} d_m T_m(x^{'}) \Bigg|_{x^{'}=\frac{2}{b-a}\left(x-\frac{a+b}{2}\right)} \tag{3.3.5}$$

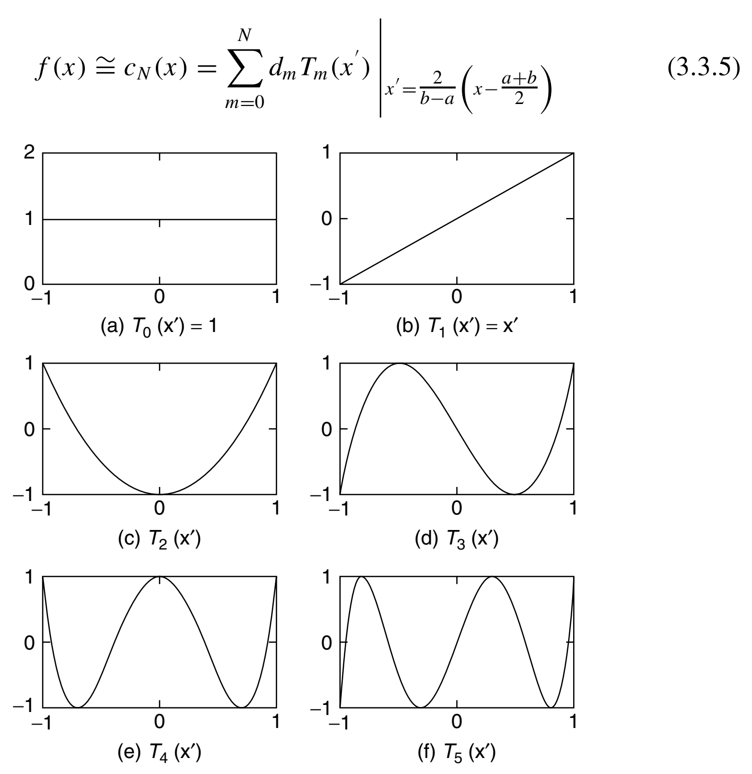


Figure 3.5 Chebyshev polynomial functions.

Table 3.3 Chebyshev Coefficient Polynomials

$T_0(x') = 1$
$T_1(x') = x'$ (x': a variable normalized onto [−1, 1])
$T_2(x') = 2x'^2 - 1$
$T_3(x') = 4x'^3 - 3x'$
$T_4(x') = 8x'^4 - 8x'^2 + 1$
$T_5(x') = 16x'^5 - 20x'^3 + 5x'$
$T_6(x') = 32x'^6 - 48x'^4 + 18x'^2 - 1$
$T_7(x') = 64x'^7 - 112x'^5 + 56x'^3 - 7x'$

where

$$d_0 = \frac{1}{N+1}\sum_{k=0}^{N} f(x_k)T_0(x_k') = \frac{1}{N+1}\sum_{k=0}^{N} f(x_k) \tag{3.3.6a}$$

$$\begin{aligned} d_m &= \frac{2}{N+1}\sum_{k=0}^{N} f(x_k)T_m(x_k') \\ &= \frac{2}{N+1}\sum_{k=0}^{N} f(x_k)\cos\frac{m(2N+1-2k)}{2(N+1)}\pi \qquad \text{for } m = 1, 2, \ldots, N \end{aligned} \tag{3.3.6b}$$

```
function [c,x,y] = cheby(f,N,a,b)
%Input : f = function name on [a,b]
%Output: c = Newton polynomial coefficients of degree N
%        (x,y) = Chebyshev nodes
if nargin == 2, a = -1; b = 1; end
k = [0: N];
theta = (2*N + 1 - 2*k)*pi/(2*N + 2);
xn = cos(theta);                        %Eq.(3.3.1a)
x = (b - a)/2*xn +(a + b)/2;            %Eq.(3.3.1b)
y = feval(f,x);
d(1) = y*ones(N + 1,1)/(N+1);
for m = 2: N + 1
   cos_mth = cos((m-1)*theta);
   d(m) = y*cos_mth'*2/(N + 1);         %Eq.(3.3.6b)
end
xn = [2  -(a + b)]/(b - a);             %the inverse of (3.3.1b)
T_0 = 1;       T_1 = xn;                %Eq.(3.3.3b)
c = d(1)*[0  T_0] +d(2)*T_1;            %Eq.(3.3.5)
for  m = 3: N + 1
   tmp = T_1;
   T_1 = 2*conv(xn,T_1) -[0  0  T_0];   %Eq.(3.3.3a)
   T_0 = tmp;
   c = [0  c] + d(m)*T_1;               %Eq.(3.3.5)
end
```

We can apply this formula to get the polynomial approximation directly for a given function $f(x)$, without having to resort to the Lagrange or Newton polynomial. Given a function, the degree of the approximate polynomial, and the left/right boundary points of the interval, the above MATLAB routine "`cheby()`" uses this formula to make the Chebyshev polynomial approximation.

The following example illustrates that this formula gives the same approximate polynomial function as could be obtained by applying the Newton polynomial with the Chebyshev nodes.

Example 3.1. Approximation by Chebyshev Polynomial. Consider the problem of finding the second-degree ($N = 2$) polynomial to approximate the function $f(x) = 1/(1 + 8x^2)$. We make the following program "`do_cheby.m`", which uses the MATLAB routine "`cheby()`" for this job and uses Lagrange/Newton polynomial with the Chebyshev nodes to do the same job. Readers can run this program to check if the results are the same.

```
%do_cheby.m
N = 2; a = -2; b = 2;
[c,x1,y1] = cheby('f31',N,a,b) %Chebyshev polynomial ftn
%for comparison with Lagrange/Newton polynomial ftn
k = [0:N]; xn = cos((2*N + 1 - 2*k)*pi/2/(N + 1));%Eq.(3.3.1a):Chebyshev nodes
x = ((b-a)*xn +a + b)/2;  %Eq.(3.3.1b)
y = f31(x); n = newtonp(x,y), l = lagranp(x,y)
```

```
>>do_cheby
    c = -0.3200   -0.0000    1.0000
```

3.4 PADE APPROXIMATION BY RATIONAL FUNCTION

Pade approximation tries to approximate a function $f(x)$ around a point x^o by a rational function

$$p_{M,N}(x - x^o) = \frac{Q_M(x - x^o)}{D_N(x - x^o)} \quad \text{with } M = N \text{ or } M = N + 1$$
$$= \frac{q_0 + q_1(x - x^o) + q_2(x - x^o)^2 + \cdots + q_M(x - x^o)^M}{1 + d_1(x - x^o) + d_2(x - x^o)^2 + \cdots + d_N(x - x^o)^N} \tag{3.4.1}$$

where $f(x^o), f'(x^o), f^{(2)}(x^o), \ldots, f^{(M+N)}(x^o)$ are known.

How do we find such a rational function? We write the Taylor series expansion of $f(x)$ up to degree $M + N$ at $x = x^o$ as

$$\begin{aligned} f(x) \approx T_{M+N}(x - x^o) &= f(x^o) + f'(x^o)(x - x^o) \\ &+ \frac{f^{(2)}(x^o)}{2}(x - x^o)^2 + \cdots + \frac{f^{(M+N)}(x^o)}{(M+N)!}(x - x^o)^{M+N} \\ &= a_0 + a_1(x - x^o) + a_2(x - x^o)^2 + \cdots + a_{M+N}(x - x^o)^{M+N} \end{aligned} \tag{3.4.2}$$

Assuming $x^o = 0$ for simplicity, we get the coefficients of $D_N(x)$ and $Q_M(x)$ such that

$$T_{M+N}(x) - \frac{Q_M(x)}{D_N(x)} = 0$$

$$\frac{(a_0 + a_1 x + \cdots + a_{M+N}x^{M+N})(1 + d_1 x + \cdots + d_N x^N) - (q_0 + q_1 x + \cdots + q_M x^M)}{1 + d_1 x + d_2 x^2 + \cdots + d_N x^N} = 0$$

$$\begin{aligned} &(a_0 + a_1 x + \cdots + a_{M+N}x^{M+N})(1 + d_1 x + \cdots + d_N x^N) \\ &\quad = q_0 + q_1 x + \cdots + q_M x^M \end{aligned} \tag{3.4.3}$$

by solving the following equations:

$$\begin{array}{llllll} a_0 & & & & & = q_0 \\ a_1 & + a_0 d_1 & & & & = q_1 \\ a_2 & + a_1 d_1 & + a_0 d_2 & & & = q_2 \\ \cdots & \cdots & \cdots & \cdots & & \cdots \\ a_M & + a_{M-1} d_1 & + a_{M-2} d_2 & \cdots & + a_{M-N} d_N & = q_M \end{array} \tag{3.4.4a}$$

$$\begin{array}{llllll} a_{M+1} & + a_M d_1 & + a_{M-1} d_2 & \cdots & + a_{M-N+1} d_N & = 0 \\ a_{M+2} & + a_{M+1} d_1 & + a_M d_2 & \cdots & + a_{M-N+2} d_N & = 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{M+N} & + a_{M+N-1} d_1 & + a_{M+N-2} d_2 & \cdots & + a_M d_N & = 0 \end{array} \tag{3.4.4b}$$

Here, we must first solve Eq. (3.4.4b) for $d_1, d_2, \ldots, d_N$ and then substitute d_i's into Eq. (3.4.4a) to obtain $q_0, q_1, \ldots, q_M$.

The MATLAB routine "`padeap()`" implements this scheme to find the coefficient vectors of the numerator/denominator polynomial $Q_M(x)/D_N(x)$ of the Pade approximation for a given function $f(x)$. Note the following things:

- The derivatives $f'(x^o), f^{(2)}(x^o), \ldots, f^{(M+N)}(x^o)$ up to order $(M+N)$ are computed numerically by using the routine "`difapx()`", that will be introduced in Section 5.3.
- In order to compute the values of the Pade approximate function, we substitute $(x - x^o)$ for x in $p_{M,N}(x)$ which has been obtained with the assumption that $x^o = 0$.

```
function [num,den] = padeap(f,xo,M,N,x0,xf)
%Input : f = function to be approximated around in [xo, xf]
%Output: num = numerator coeffs of Pade approximation of degree M
%        den = denominator coeffs of Pade approximation of degree N
a(1) = feval(f,xo);
h = .01; tmp = 1;
for i = 1:M + N
   tmp = tmp*i*h; %i!h^i
   dix = difapx(i,[-i i])*feval(f,xo+[-i:i]*h)'; %derivative(Section 5.3)
   a(i + 1) = dix/tmp; %Taylor series coefficient
end
for m = 1:N
   n = 1:N; A(m,n) = a(M + 1 + m - n);
   b(m) = -a(M + 1 + m);
end
d = A\b'; %Eq.(3.4.4b)
for m = 1: M + 1
   mm = min(m - 1,N);
   q(m) = a(m:-1:m - mm)*[1; d(1:mm)]; %Eq.(3.4.4a)
end
num = q(M + 1:-1:1)/d(N); den = [d(N:-1:1)' 1]/d(N); %descending order
if nargout == 0 % plot the true ftn, Pade ftn and Taylor expansion
   if nargin < 6, x0 = xo - 1; xf = xo + 1; end
   x = x0+[xf-x0]/100*[0:100]; yt = feval(f,x);
   x1 = x-xo; yp = polyval(num,x1)./polyval(den,x1);
   yT = polyval(a(M + N + 1:-1:1),x1);
   clf, plot(x,yt,'k', x,yp,'r', x,yT,'b')
end
```

Example 3.2. Pade Approximation for $f(x) = e^x$. Let's find the Pade approximation $p_{3,2}(x) = Q_3(x)/D_2(x)$ for $f(x) = e^x$ around $x^o = 0$. We make the MATLAB program "`do_pade.m`", which uses the routine "`padeap()`" for this job and uses it again with no output argument to see the graphic results as depicted in Fig. 3.6.

```
>>do_pade %Pade approximation
  n = 0.3333   2.9996   11.9994   19.9988
  d = 1.0000  -7.9997   19.9988
```

```
%do_pade.m to get the Pade approximation for f(x) = e^x
f1 = inline('exp(x)','x');
M = 3; N = 2; %the degrees of Numerator Q(x) and Denominator D(x)
xo = 0; %the center of Taylor series expansion
[n,d] = padeap(f1,xo,M,N) %to get the coefficients of Q(x)/P(x)
x0 = -3.5; xf = 0.5; %left/right boundary of the interval
padeap(f1,xo,M,N,x0,xf) %to see the graphic results
```

To confirm and support this result from the analytical point of view and to help the readers understand the internal mechanism, we perform the hand-calculation

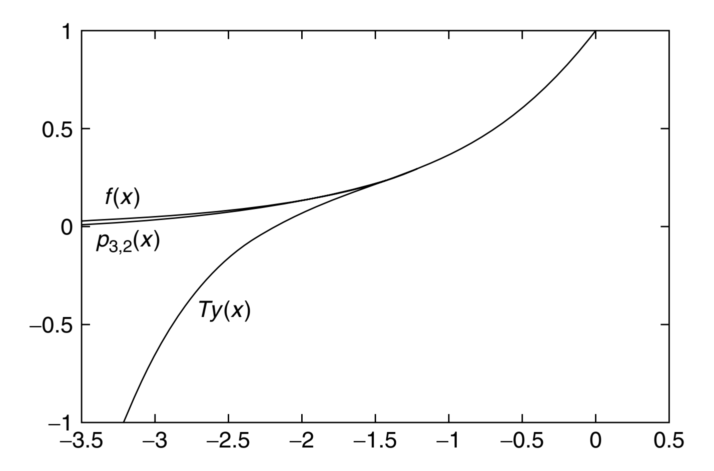


Figure 3.6 Pade approximation and Taylor series expansion for $f(x) = e^x$(Example 3.2.).

procedure. First, we write the Taylor series expansion at $x = 0$ up to degree $M + N = 5$ for the given function $f(x) = e^x$ as

$$Ty(x) = \sum_{k=0}^{M+N} \frac{f^{(k)}(x)}{k!}x^k = 1 + x + \frac{1}{2}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 + \frac{1}{5!}x^5 + \cdots \tag{E3.2.1}$$

whose coefficients are

$$a_0 = 1, \quad a_1 = 1, \quad a_2 = \frac{1}{2}, \quad a_3 = \frac{1}{6}, \quad a_4 = \frac{1}{24}, \quad a_5 = \frac{1}{120}, \ldots \tag{E3.2.2}$$

We put this into Eq. (3.4.4b) with $M = 3, N = 2$ and solve it for d_i's to get $D_2(x) = 1 + d_1x + d_2x^2$.

$$\begin{matrix} a_4 + a_3d_1 + a_2d_2 = 0 \\ a_3 + a_2d_1 + a_1d_2 = 0 \end{matrix}, \begin{bmatrix} 1/6 & 1/2 \\ 1/24 & 1/6 \end{bmatrix}\begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} -1/24 \\ -1/120 \end{bmatrix}, \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} -2/5 \\ a1/20 \end{bmatrix} \tag{E3.2.3}$$

Substituting this to Eq. (3.4.4a) yields

$$\begin{aligned} q_0 &= a_0 = 1 \\ q_1 &= a_1 + a_0d_1 = 1 + 1 \times (-2/5) = 3/5 \\ q_2 &= a_2 + a_1d_1 + a_0d_2 = 1/2 + 1 \times (-2/5) + 1 \times (1/20) = 3/20 \\ q_3 &= a_3 + a_2d_1 + a_1d_2 = 1/6 + (1/2) \times (-2/5) + 1 \times (1/20) = 1/60 \end{aligned} \tag{E3.2.4}$$

With these coefficients, we write the Pade approximate function as

$$p_{3,2}(x) = \frac{Q_3(x)}{D_2(x)} = \frac{1 + (3/5)x + (3/20)x^2 + (1/60)x^3}{1 + (-2/5)x + (1/20)x^2}$$
$$= \frac{(1/3)x^3 + 3x^2 + 12x + 20}{x^2 - 8x + 20} \quad \text{(E3.2.5)}$$

3.5 INTERPOLATION BY CUBIC SPLINE

If we use the Lagrange/Newton polynomial to interpolate a given set of $N + 1$ data points, the polynomial is usually of degree N and so has $N - 1$ local extrema (maxima/minima). Thus, it will show a wild swing/oscillation (called 'polynomial wiggle'), particularly near the ends of the whole interval as the number of data points increases and so the degree of the polynomial gets higher, as illustrated in Fig. 3.2. Then, how about a piecewise-linear approach, like assigning the individual approximate polynomial to every subinterval between data points? How about just a linear interpolation—that is, connecting the data points by a straight line? It is so simple, but too short of smoothness. Even with the second-degree polynomial, the piecewise-quadratic curve is not smooth enough to please our eyes, since the second-order derivatives of quadratic polynomials for adjacent subintervals can't be made to conform with each other. In real life, there are many cases where the continuity of second-order derivatives is desirable. For example, it is very important to ensure the smoothness up to order 2 for interpolation needed in CAD (computer-aided design)/CAM (computer-aided manufacturing), computer graphic, and robot path/trajectory planning. That's why we often resort to the piecewise-cubic curve constructed by the individual third-degree polynomials assigned to each subinterval, which is called the cubic spline interpolation. (A spline is a kind of template that architects use to draw a smooth curve between two points.)

For a given set of data points $\{(x_k, y_k), k = 0 : N\}$, the cubic spline $s(x)$ consists of N cubic polynomial $s_k(x)$'s assigned to each subinterval satisfying the following constraints (S0)–(S4).

(S0) $s(x) = s_k(x) = S_{k,3}(x - x_k)^3 + S_{k,2}(x - x_k)^2 + S_{k,1}(x - x_k) + S_{k,0}$ for $x \in [x_k, x_{k+1}], k = 0 : N$

(S1) $s_k(x_k) = S_{k,0} = y_k$ for $k = 0 : N$

(S2) $s_{k-1}(x_k) \equiv s_k(x_k) = S_{k,0} = y_k$ for $k = 1 : N - 1$

(S3) $s'_{k-1}(x_k) \equiv s'_k(x_k) = S_{k,1}$ for $k = 1 : N - 1$

(S4) $s''_{k-1}(x_k) \equiv s''_k(x_k) = 2S_{k,2}$ for $k = 1 : N - 1$

These constraints (S1)–(S4) amount to a set of $N + 1 + 3(N - 1) = 4N - 2$ linear equations having $4N$ coefficients of the N cubic polynomials

$$\{S_{k,0}, S_{k,1}, S_{k,2}, S_{k,3}, k = 0 : N - 1\}$$

Table 3.4 Boundary Conditions for a Cubic Spline

(i)	First-order derivatives specified	$s'_0(x_0) = S_{0,1},\ s'_N(x_N) = S_{N,1}$
(ii)	Second-order derivatives specified (end-curvature adjusted)	$s''_0(x_0) = 2S_{0,2},\ s''_N(x_N) = 2S_{N,2}$
(iii)	Second-order derivatives extrapolated	$s''_0(x_0) \equiv s''_1(x_1) + \frac{h_0}{h_1}(s''_1(x_1) - s''_2(x_2))$ $s''_N(x_N) \equiv s''_{N-1}(x_{N-1}) + \frac{h_{N-1}}{h_{N-2}}(s''_{N-1}(x_{N-1}) - s''_{N-2}(x_{N-2}))$

as their unknowns. Two additional equations necessary for the equations to be solvable are supposed to come from the boundary conditions for the first/second-order derivatives at the end points (x_0, y_0) and (x_N, y_N) as listed in Table 3.4.

Now, noting from (S1) that $S_{k,0} = y_k$, we will arrange the constraints (S2)–(S4) and eliminate $S_{k,1}, S_{k,3}$'s to set up a set of equations with respect to the $N + 1$ unknowns $\{S_{k,2}, k = 0 : N\}$. In order to do so, we denote each interval width by $h_k = x_{k+1} - x_k$ and substitute (S0) into (S4) to write

$$s''_k(x_{k+1}) = 6S_{k,3}h_k + 2S_{k,2} \equiv s''_{k+1}(x_{k+1}) = 2S_{k+1,2}$$

$$S_{k,3}h_k = \frac{1}{3}(S_{k+1,2} - S_{k,2}) \tag{3.5.1a}$$

$$S_{k-1,3}h_{k-1} = \frac{1}{3}(S_{k,2} - S_{k-1,2}) \tag{3.5.1b}$$

We substitute these equations into (S2) with $k + 1$ in place of k

$$s_k(x_{k+1}) = S_{k,3}(x_{k+1} - x_k)^3 + S_{k,2}(x_{k+1} - x_k)^2 + S_{k,1}(x_{k+1} - x_k) + S_{k,0} \equiv y_{k+1}$$

$$S_{k,3}h_k^3 + S_{k,2}h_k^2 + S_{k,1}h_k + y_k \equiv y_{k+1}$$

to eliminate $S_{k,3}$'s and rewrite it as

$$\frac{h_k}{3}(S_{k+1,2} - S_{k,2}) + S_{k,2}h_k + S_{k,1} = \frac{y_{k+1} - y_k}{h_k} = dy_k$$

$$h_k(S_{k+1,2} + 2S_{k,2}) + 3S_{k,1} = 3\,dy_k \tag{3.5.2a}$$

$$h_{k-1}(S_{k,2} + 2S_{k-1,2}) + 3S_{k-1,1} = 3\,dy_{k-1} \tag{3.5.2b}$$

We also substitute Eq. (3.5.1b) into (S3)

$$s'_{k-1}(x_k) = 3S_{k-1,3}h_{k-1}^2 + 2S_{k-1,2}h_{k-1} + S_{k-1,1} \equiv s'_k(x_k) = S_{k,1}$$

to write

$$S_{k,1} - S_{k-1,1} = h_{k-1}(S_{k,2} - S_{k-1,2}) + 2h_{k-1}S_{k-1,2} = h_{k-1}(S_{k,2} + S_{k-1,2}) \tag{3.5.3}$$

In order to use this for eliminating $S_{k,1}$ from Eq. (3.5.2), we subtract (3.5.2b) from (3.5.2a) to write

$$h_k(S_{k+1,2} + 2S_{k,2}) - h_{k-1}(S_{k,2} + 2S_{k-1,2}) + 3(S_{k,1} - S_{k-1,1}) = 3(dy_k - dy_{k-1})$$

and then substitute Eq. (3.5.3) into this to write

$$\begin{aligned} &h_k(S_{k+1,2} + 2S_{k,2}) - h_{k-1}(S_{k,2} + 2S_{k-1,2}) + 3h_{k-1}(S_{k,2} + S_{k-1,2}) \\ &\quad = 3(dy_k - dy_{k-1}) \\ &h_{k-1}S_{k-1,2} + 2(h_{k-1} + h_k)S_{k,2} + h_kS_{k+1,2} = 3(dy_k - dy_{k-1}) \qquad (3.5.4) \\ &\qquad\qquad \text{for } k = 1 : N - 1 \end{aligned}$$

Since these are $N - 1$ equations with respect to $N + 1$ unknowns $\{S_{k,2}, k = 0 : N\}$, we need two more equations from the boundary conditions to be given as listed in Table 3.4.

How do we convert the boundary condition into equations? In the case where the first-order derivatives on the two boundary points are given as (i) in Table 3.4, we write Eq. (3.5.2a) for $k = 0$ as

$$h_0(S_{1,2} + 2S_{0,2}) + 3S_{0,1} = 3\,dy_0, \qquad 2h_0S_{0,2} + h_0S_{1,2} = 3(dy_0 - S_{0,1}) \tag{3.5.5a}$$

We also write Eq. (3.5.2b) for $k = N$ as

$$h_{N-1}(S_{N,2} + 2S_{N-1,2}) + 3S_{N-1,1} = 3\,dy_{N-1}$$

and substitute (3.5.3)($k = N$) into this to write

$$\begin{aligned} h_{N-1}(S_{N,2} + 2S_{N-1,2}) + 3S_{N,1} - 3h_{N-1}(S_{N,2} + S_{N-1,2}) &= 3\,dy_{N-1} \\ h_{N-1}S_{N-1,2} + 2h_{N-1}S_{N,2} &= 3(S_{N,1} - dy_{N-1}) \qquad (3.5.5b) \end{aligned}$$

Equations (3.5.5a) and (3.5.5b) are two additional equations that we need to solve Eq. (3.5.4) and that's it. In the case where the second-order derivatives on the two boundary points are given as (ii) in Table 3.4, $S_{0,2}$ and $S_{N,2}$ are directly known from the boundary conditions as

$$S_{0,2} = s_0''(x_0)/2, \qquad S_{N,2} = s_N''(x_N)/2 \tag{3.5.6}$$

and, subsequently, we have just $N-1$ unknowns. In the case where the second-order derivatives on the two boundary points are given as (iii) in Table 3.4

$$s_0''(x_0) \equiv s_1''(x_1) + \frac{h_0}{h_1}(s_1''(x_1) - s_2''(x_2))$$

$$s_N''(x_N) \equiv s_{N-1}''(x_{N-1}) + \frac{h_{N-1}}{h_{N-2}}(s_{N-1}''(x_{N-1}) - s_{N-2}''(x_{N-2}))$$

we can instantly convert these into two equations with respect to $S_{0,2}$ and $S_{N,2}$ as

$$h_1 S_{0,2} - (h_0 + h_1)S_{1,2} + h_0 S_{2,2} = 0 \qquad (3.5.7a)$$

$$h_{N-2} S_{N,2} - (h_{N-1} + h_{N-2})S_{N-1,2} + h_{N-1} S_{N-2,2} = 0 \qquad (3.5.7b)$$

Finally, we combine the two equations (3.5.5a) and (3.5.5b) with Eq. (3.5.4) to write it in the matrix–vector form as

$$\begin{bmatrix} 2h_0 & h_0 & 0 & \bullet & \bullet \\ h_0 & 2(h_0+h_1) & h_1 & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet & 0 \\ \bullet & \bullet & h_{N-2} & 2(h_{N-2}+h_{N-1}) & h_{N-1} \\ \bullet & \bullet & 0 & h_{N-1} & 2h_{N-1} \end{bmatrix} \begin{bmatrix} S_{0,2} \\ S_{1,2} \\ \bullet \\ S_{N-1,2} \\ S_{N,2} \end{bmatrix}$$

$$= \begin{bmatrix} 3(dy_0 - S_{0,1}) \\ 3(dy_1 - dy_0) \\ \bullet \\ 3(dy_{N-1} - dy_{N-2}) \\ 3(S_{N,1} - dy_{N-1}) \end{bmatrix} \qquad (3.5.8)$$

After solving this system of equation for $\{S_{k,2}, k = 0 : N\}$, we substitute them into (S1), (3.5.2), and (3.5.1) to get the other coefficients of the cubic spline as

$$S_{k,0} \overset{(S1)}{=} y_k, \quad S_{k,1} \overset{(3.5.2)}{=} dy_k - \frac{h_k}{3}(S_{k+1,2} + 2S_{k,2}), \quad S_{k,3} \overset{(3.5.1)}{=} \frac{S_{k+1,2} - S_{k,2}}{3h_k} \qquad (3.5.9)$$

The MATLAB routine "`cspline()`" constructs Eq.(3.5.8), solves it to get the cubic spline coefficients for given x, y coordinates of the data points and the boundary conditions, uses the `mkpp()` routine to get the piecewise polynomial expression, and then uses the `ppval()` routine to obtain the value(s) of the piecewise polynomial function for `xi`—that is, the interpolation over `xi`. The type of the boundary condition is supposed to be specified by the third input argument `KC`. In the case where the boundary condition is given as (i)/(ii) in Table 3.4, the input argument `KC` should be set to 1/2 and the fourth and fifth input arguments must be the first/second derivatives at the end points. In the case where the boundary condition is given as extrapolated like (iii) in Table 3.4, the input argument `KC` should be set to 3 and the fourth and fifth input arguments do not need to be fed.

```
function  [yi,S] = cspline(x,y,xi,KC,dy0,dyN)
%This function finds the cubic splines for the input data points (x,y)
%Input:  x = [x0 x1 ... xN], y = [y0 y1 ... yN], xi=interpolation points
%        KC = 1/2 for 1st/2nd derivatives on boundary specified
%        KC = 3 for 2nd derivative on boundary extrapolated
%        dy0 = S'(x0) = S01: initial derivative
%        dyN = S'(xN) = SN1: final derivative
%Output: S(n,k); n = 1:N, k = 1,4 in descending order
if nargin < 6, dyN = 0; end, if nargin < 5, dy0 = 0; end
if nargin < 4,  KC = 0; end
N = length(x) - 1;
% constructs a set of equations w.r.t. {S(n,2), n = 1:N + 1}
A = zeros(N + 1,N + 1); b = zeros(N + 1,1);
S = zeros(N + 1,4);    % Cubic spline coefficient matrix
k = 1:N; h(k) = x(k + 1) - x(k); dy(k) = (y(k + 1) - y(k))/h(k);
% Boundary condition
if KC <= 1  %1st derivatives specified
  A(1,1:2) = [2*h(1) h(1)]; b(1) = 3*(dy(1) - dy0); %Eq.(3.5.5a)
  A(N + 1,N:N + 1) = [h(N) 2*h(N)]; b(N + 1) = 3*(dyN - dy(N));%Eq.(3.5.5b)
 elseif KC == 2  %2nd derivatives specified
  A(1,1) = 2; b(1) = dy0; A(N + 1,N+1) = 2; b(N + 1) = dyN; %Eq.(3.5.6)
 else %2nd derivatives extrapolated
  A(1,1:3) =  [h(2) - h(1) - h(2) h(1)]; %Eq.(3.5.7)
  A(N + 1,N-1:N + 1) = [h(N) - h(N)-h(N - 1) h(N - 1)];
end
for m = 2:N  %Eq.(3.5.8)
  A(m,m - 1:m + 1) = [h(m - 1) 2*(h(m - 1) + h(m)) h(m)];
  b(m) = 3*(dy(m) - dy(m - 1));
end
S(:,3) = A\b;
% Cubic spline coefficients
for m = 1: N
  S(m,4) = (S(m+1,3)-S(m,3))/3/h(m);  %Eq.(3.5.9)
  S(m,2) = dy(m) -h(m)/3*(S(m + 1,3)+2*S(m,3));
  S(m,1) = y(m);
end
S = S(1:N, 4:-1:1); %descending order
pp = mkpp(x,S); %make piecewise polynomial
yi = ppval(pp,xi); %values of piecewise polynomial ftn
```

(cf) See Problem 1.11 for the usages of the MATLAB routines `mkpp()` and `ppval()`.

Example 3.3. Cubic Spline. Consider the problem of finding the cubic spline interpolation for the $N + 1 = 4$ data points

$$\{(0, 0), (1, 1), (2, 4), (3, 5)\} \tag{E3.3.1}$$

subject to the boundary condition

$$s_0'(x_0) = s_0'(0) = S_{0,1} = 2, \quad s_N'(x_N) = h_3'(3) = h_{3,1} = 2 \tag{E3.3.2}$$

With the subinterval widths on the x-axis and the first divided differences as

$$h_0 = h_1 = h_2 = h_3 = 1$$

$$dy_0 = \frac{y_1 - y_0}{h_0} = 1, \quad dy_1 = \frac{y_2 - y_1}{h_1} = 3, \quad dy_2 = \frac{y_3 - y_2}{h_2} = 1 \tag{E3.3.3}$$

we write Eq. (3.5.8) as

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} S_{0,2} \\ S_{1,2} \\ S_{2,2} \\ S_{3,2} \end{bmatrix} = \begin{bmatrix} 3(dy_0 - S_{0,1}) \\ 3(dy_1 - dy_0) \\ 3(dy_2 - dy_1) \\ 3(S_{3,1} - dy_1) \end{bmatrix} = \begin{bmatrix} -3 \\ 6 \\ -6 \\ 3 \end{bmatrix} \tag{E3.3.4}$$

Then we solve this equation to get

$$S_{0,2} = -3, \qquad S_{1,2} = 3, \qquad S_{2,2} = -3, \qquad S_{3,2} = 3 \tag{E3.3.5}$$

and substitute this into Eq. (3.5.9) to obtain

$$S_{0,0} = 0, \qquad S_{1,0} = 1, \qquad S_{2,0} = 4 \tag{E3.3.6}$$

$$S_{0,1} = dy_0 - \frac{h_0}{3}(S_{1,2} + 2S_{0,2}) = 1 - \frac{1}{3}(3 + 2 \times (-3)) = 2 \tag{E3.3.7a}$$

$$S_{1,1} = dy_1 - \frac{h_1}{3}(S_{2,2} + 2S_{1,2}) = 3 - \frac{1}{3}(-3 + 2 \times 3) = 2 \tag{E3.3.7b}$$

$$S_{2,1} = dy_2 - \frac{h_2}{3}(S_{3,2} + 2S_{2,2}) = 1 - \frac{1}{3}(3 + 2 \times (-3)) = 2 \tag{E3.3.7c}$$

$$S_{0,3} = \frac{S_{1,2} - S_{0,2}}{3h_0} = \frac{3 - (-3)}{3} = 2 \tag{E3.3.8a}$$

$$S_{1,3} = \frac{S_{2,2} - S_{1,2}}{3h_1} = \frac{-3 - 3}{3} = -2 \tag{E3.3.8b}$$

$$S_{2,3} = \frac{S_{3,2} - S_{2,2}}{3h_2} = \frac{3 - (-3)}{3} = 2 \tag{E3.3.8c}$$

```
%do_csplines.m
KC = 1; dy0 = 2; dyN = 2; % with specified 1st derivatives on boundary
x = [0  1  2  3];  y = [0  1  4  5];
xi = x(1)+[0:200]*(x(end)-x(1))/200; %intermediate points
[yi,S] = cspline(x,y,xi,KC,dy0,dyN); S %cubic spline interpolation
clf, plot(x,y,'ko',xi,yi,'k:')
yi = spline(x,[dy0 y dyN],xi); %for comparison with MATLAB built-in ftn
hold on, pause, plot(x,y,'ro',xi,yi,'r:')
yi = spline(x,y,xi); %for comparison with MATLAB built-in ftn
pause, plot(x,y,'bo',xi,yi,'b')
KC = 3; [yi,S] = cspline(x,y,xi,KC);%with the 2nd derivatives extrapolated
pause, plot(x,y,'ko',xi,yi,'k')
```

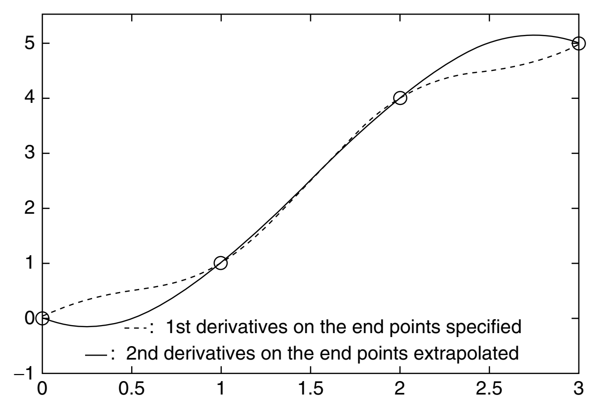


Figure 3.7 Cubic splines for Example 3.3.

Finally, we can write the cubic spline equations collectively from (S0) as

$$\begin{aligned}
s_0(x) &= S_{0,3}(x-x_0)^3 + S_{0,2}(x-x_0)^2 + S_{0,1}(x-x_0) + S_{0,0} \\
&= 2x^3 - 3x^2 + 2x + 0 \\
s_1(x) &= S_{1,3}(x-x_1)^3 + S_{1,2}(x-x_1)^2 + S_{1,1}(x-x_1) + S_{1,0} \\
&= -2(x-1)^3 + 3(x-1)^2 + 2(x-1) + 1 \\
s_2(x) &= S_{2,3}(x-x_2)^3 + S_{2,2}(x-x_2)^2 + S_{2,1}(x-x_2) + S_{2,0} \\
&= 2(x-2)^3 - 3(x-2)^2 + 2(x-1) + 4
\end{aligned}$$

We make and run the program "`do_csplines.m`", which uses the routine "`cspline()`" to compute the cubic spline coefficients $\{S_{k,3}, S_{k,2}, S_{k,1}, S_{k,0}, k = 0 : N-1\}$ and obtain the value(s) of the cubic spline function for `xi` (i.e., the interpolation over `xi`) and then plots the result as depicted in Fig. 3.7. We also compare this result with that obtained by using the MATLAB built-in function "`spline(x,y,xi)`", which works with the boundary condition of type (i) for the second input argument given as `[dy0 y dyN]`, and with the boundary condition of type (iii) for the same lengths of `x` and `y`.

```
>>do_csplines %cubic spline
  S = 2.0000  -3.0000   2.0000      0
     -2.0000   3.0000   2.0000 1.0000
      2.0000  -3.0000   2.0000 4.0000
```

3.6 HERMITE INTERPOLATING POLYNOMIAL

In some cases, we need to find the polynomial function that not only passes through the given points, but also has the specified derivatives at every data point. We call such a polynomial the Hermite interpolating polynomial or the osculating polynomial.

For simplicity, we consider a third-order polynomial

$$h(x) = H_3x^3 + H_2x^2 + H_1x + H_0 \tag{3.6.1}$$

matching just two points (x_0, y_0), (x_1, y_1) and having the specified first derivatives y'_0, y'_1 at the points. We can obtain the four coefficients H_3, H_2, H_1, H_0 by solving

$$\begin{aligned}
h(x_0) &= H_3x_0^3 + H_2x_0^2 + H_1x_0 + H_0 = y_0 \\
h(x_1) &= H_3x_1^3 + H_2x_1^2 + H_1x_1 + H_0 = y_1 \\
h'(x_0) &= 3H_3x_0^2 + 2H_2x_0 + H_1 = y'_0 \\
h'(x_1) &= 3H_3x_1^2 + 2H_2x_1 + H_1 = y'_1
\end{aligned} \tag{3.6.2}$$

As an alternative, we approximate the specified derivatives at the data points by their differences

$$y'_0 = \frac{h(x_0 + \varepsilon) - h(x_0)}{\varepsilon} = \frac{y_2 - y_0}{\varepsilon}, \qquad y'_1 = \frac{h(x_1) - h(x_1 - \varepsilon)}{\varepsilon} = \frac{y_1 - y_3}{\varepsilon} \tag{3.6.3}$$

and find the Lagrange/Newton polynomial matching the four points

$$(x_0, y_0),\ (x_2 = x_0 + \varepsilon, y_2 = y_0 + y'_0\varepsilon), (x_3 = x_1 - \varepsilon, y_3 = y_1 - y'_1\varepsilon),\ (x_1, y_1) \tag{3.6.4}$$

The MATLAB routine "`hermit()`" constructs Eq. (3.6.2) and solves it to get the Hermite interpolating polynomial coefficients for a single interval given the two end points and the derivatives at them as the input arguments. The next routine "`hermits()`" uses "`hermit()`" to get the Hermite coefficients for a set of multiple subintervals.

```
function H = hermit(x0,y0,dy0,x1,y1,dy1)
A = [x0^3    x0^2   x0   1;   x1^3    x1^2   x1   1;
     3*x0^2  2*x0   1    0;   3*x1^2  2*x1   1    0];
b = [y0  y1 dy0 dy1]'; %Eq.(3.6-2)
H = (A\b)';
```

```
function H = hermits(x,y,dy)
% finds Hermite interpolating polynomials for multiple subintervals
%Input : [x,y],dy - points and derivatives at the points
%Output: H = coefficients of cubic Hermite interpolating polynomials
for n = 1:length(x)-1
   H(n,:) = hermit(0,y(n),dy(n),x(n + 1)-x(n),y(n + 1),dy(n + 1));
end
```

Example 3.4. Hermite Interpolating Polynomial. Consider the problem of finding the polynomial interpolation for the $N + 1 = 4$ data points

$$\{(0, 0), (1, 1), (2, 4), (3, 5)\} \tag{E3.4.1}$$

subject to the conditions

$$h'_0(x_0) = h'_0(0) = 2, \quad h'_1(1) = 0, \quad h'_2(2) = 0, \quad h'_N(x_N) = h'_3(3) = 2 \tag{E3.4.2}$$

For this problem, we only have to type the following statements in the MATLAB command window.

```
>>x = [0 1 2 3]; y = [0 1 4 5]; dy = [2 0 0 2]; xi = [0:0.01:3];
>>H = hermits(x,y,dy); yi = ppval(mkpp(x,H), xi);
```

3.7 TWO-DIMENSIONAL INTERPOLATION

In this section we deal with only the simplest way of two-dimensional interpolation—that is, a generalization of piecewise linear interpolation called

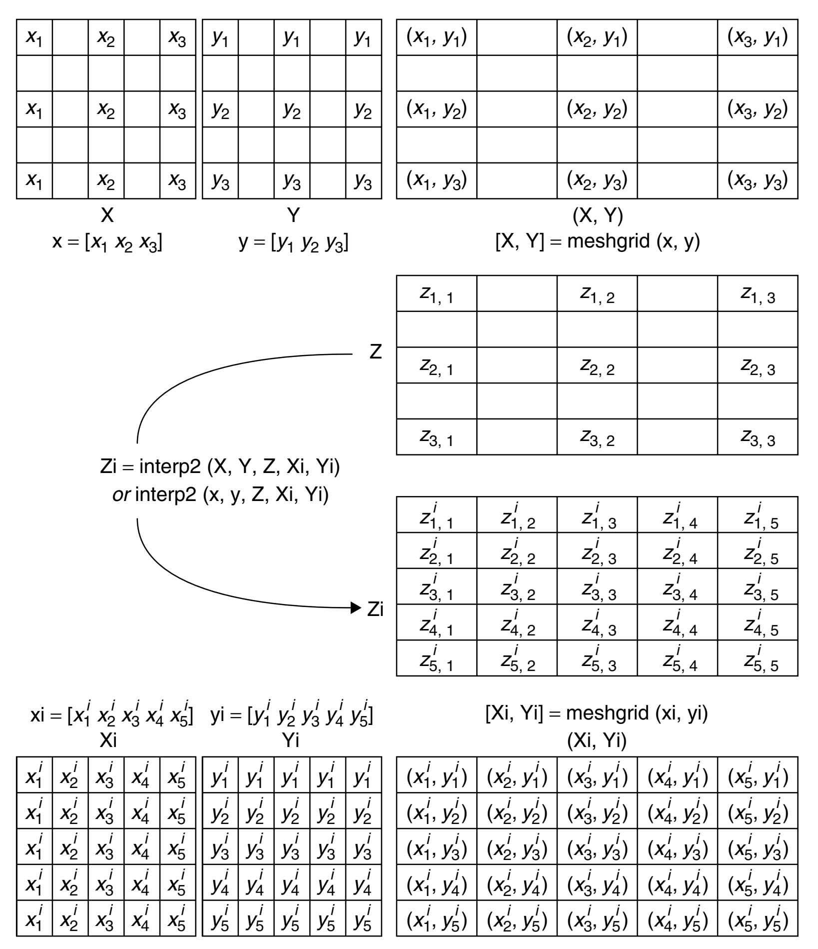


Figure 3.8 A two-dimensional interpolation using `Zi = interp2()` on the grid points `[Xi,Yi]` generated by the `meshgrid()` command.

the bilinear interpolation. The bilinear interpolation for a point (x, y) on the rectangular sub-region having (x_{m-1}, y_{n-1}) and (x_m, y_n) as its left-upper/right-lower corner points is described by the following formula.

$$z(x, y_{n-1}) = \frac{x_m - x}{x_m - x_{m-1}} z_{m-1,n-1} + \frac{x - x_{m-1}}{x_m - x_{m-1}} z_{m,n-1} \tag{3.7.1a}$$

$$z(x, y_n) = \frac{x_m - x}{x_m - x_{m-1}} z_{m-1,n} + \frac{x - x_{m-1}}{x_m - x_{m-1}} z_{m,n} \tag{3.7.1b}$$

$$\begin{aligned} z(x, y) &= \frac{y_n - y}{y_n - y_{n-1}} z(x, y_{n-1}) + \frac{y - y_{n-1}}{y_n - y_{n-1}} z(x, y_n) \\ &= \frac{1}{(x_m - x_{m-1})(y_n - y_{n-1})} \{(x_m - x)(y_n - y) z_{m-1,n-1} \\ &\quad + (x - x_{m-1})(y_n - y) z_{m,n-1} + (x_m - x)(y - y_{n-1}) z_{m-1,n} \\ &\quad + (x - x_{m-1})(y - y_{n-1}) z_{m,n}\} \quad \text{for } x_{m-1} \le x \le x_m,\ y_{n-1} \le y \le y_n \end{aligned} \tag{3.7.2}$$

```
function Zi = intrp2(x,y,Z,xi,yi)
%To interpolate Z(x,y) on (xi,yi)
M = length(x); N = length(y);
Mi = length(xi); Ni = length(yi);
for mi = 1:Mi
 for ni = 1:Ni
  for m = 2:M
   for n = 2:N
     break1 = 0;
     if xi(mi) <= x(m) & yi(ni) <= y(n)
       tmp = (x(m)-xi(mi))*(y(n)-yi(ni))*Z(n - 1,m - 1)...
              +(xi(mi) - x(m-1))*(y(n) - yi(ni))*Z(n - 1,m)...
              +(x(m) - xi(mi))*(yi(ni) - y(n - 1))*Z(n,m - 1)...
              +(xi(m) - x(m-1))*(yi(ni) - y(n-1))*Z(n,m);
       Zi(ni,mi) = tmp/(x(m) - x(m-1))/(y(n) - y(n-1)); %Eq.(3.7.2)
       break1 = 1;
     end
     if break1 > 0  break,  end
   end
   if break1 > 0  break,  end
  end
 end
end
```

This formula is cast into the MATLAB routine “`intrp2()`”, which is so named in order to distinguish it from the MATLAB built-in routine “`interp2()`”. Note that in reference to Fig. 3.8, the given values of data at grid points `(x(m),y(n))` and the interpolated values for intermediate points `(xi(m),yi(n))` are stored in `Z(n,m)` and `Zi(n,m)`, respectively.

```
%do_interp2.m
% 2-dimensional interpolation for Ex 3.5
xi = -2:0.1:2;  yi = -2:0.1:2;
[Xi,Yi] = meshgrid(xi,yi);
Z0 = Xi.^2 + Yi.^2; %(E3.5.1)
subplot(131), mesh(Xi,Yi,Z0)
x = -2:0.5:2;  y = -2:0.5:2;
[X,Y] = meshgrid(x,y);
Z = X.^2 + Y.^2;
subplot(132), mesh(X,Y,Z)
Zi = interp2(x,y,Z,Xi,Yi); %built-in routine
subplot(133), mesh(xi,yi,Zi)
Zi = intrp2(x,y,Z,xi,yi); %our own routine
pause, mesh(xi,yi,Zi)
norm(Z0 - Zi)/norm(Z0)
```

Example 3.5. Two-Dimensional Bilinear Interpolation. We consider interpolating the sample values of a function

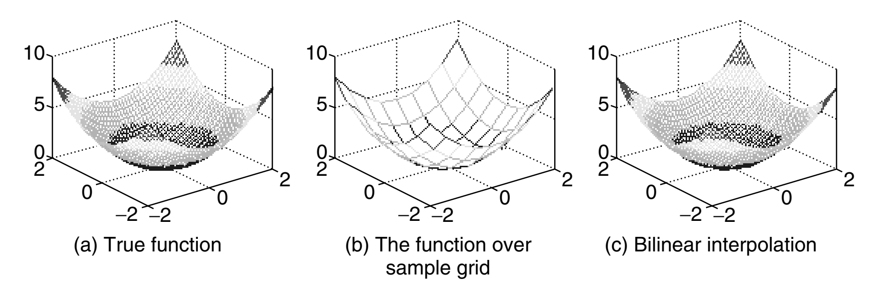
$$f(x, y) = x^2 + y^2 \tag{E3.5.1}$$

for the 5×5 grid over the 21×21 grid on the domain $D = \{(x, y) | -2 \le x \le 2, -2 \le y \le 2\}$.

We make the MATLAB program "`do_interp2.m`", which uses the routine "`intrp2()`" to do this job, compares its function with that of the MATLAB built-in routine "`interp2()`", and computes a kind of relative error to estimate how close the interpolated values are to the original values. The graphic results of running this program are depicted in Fig. 3.9, which shows that we obtained a reasonable approximation with the error of 2.6% from less than 1/16 of the original data. It is implied that the sampling may be a simple data compression method, as long as the interpolated data are little impaired.

3.8 CURVE FITTING

When many sample data pairs $\{(x_k, y_k), k = 0 : M\}$ are available, we often need to grasp the relationship between the two variables or to describe the trend of the



(a) True function (b) The function over sample grid (c) Bilinear interpolation

Figure 3.9 Two-dimensional interpolation (Example 3.5).

data, hopefully in a form of function $y = f(x)$. But, as mentioned in Remark 3.1, the polynomial approach meets with the polynomial wiggle and/or Runge phenomenon, which makes it not attractive for approximation purpose. Although the cubic spline approach may be a roundabout toward the smoothness as explained in Section 3.5, it has too many parameters and so does not seem to be an efficient way of describing the relationship or the trend, since every subinterval needs four coefficients. What other choices do we have? Noting that many data are susceptible to some error, we don't have to try to find a function passing exactly through every point. Instead of pursuing the exact matching at every data point, we look for an approximate function (not necessarily a polynomial) that describes the data points as a whole with the smallest error in some sense, which is called the curve fitting.

As a reasonable means, we consider the least-squares (LS) approach to minimizing the sum of squared errors, where the error is described by the vertical distance to the curve from the data points. We will look over various types of fitting functions in this section.

3.8.1 Straight Line Fit: A Polynomial Function of First Degree

If there is some theoretical basis on which we believe the relationship between the two variables to be

$$\theta_1 x + \theta_0 = y \tag{3.8.1}$$

we should set up the following system of equations from the collection of many experimental data:

$$\begin{aligned} \theta_1 x_1 + \theta_0 &= y_1 \\ \theta_1 x_2 + \theta_0 &= y_2 \\ &\cdots\cdots\cdots \\ \theta_1 x_M + \theta_0 &= y_M \end{aligned}$$

$$A\boldsymbol{\theta} = \mathbf{y} \qquad \text{with } A = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \cdot & \cdot \\ x_M & 1 \end{bmatrix}, \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_0 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ y_M \end{bmatrix} \tag{3.8.2}$$

Noting that this apparently corresponds to the overdetermined case mentioned in Section 2.1.3, we resort to the least-squares (LS) solution (2.1.10)

$$\boldsymbol{\theta}^{o} = \begin{bmatrix} \theta_1^{o} \\ \theta_0^{o} \end{bmatrix} = [A^T A]^{-1} A^T \mathbf{y} \tag{3.8.3}$$

which minimizes the objective function

$$J = \|\mathbf{e}\|^2 = \|A\boldsymbol{\theta} - \mathbf{y}\|^2 = [A\boldsymbol{\theta} - \mathbf{y}]^T [A\boldsymbol{\theta} - \mathbf{y}] \tag{3.8.4}$$

Sometimes we have the information about the error bounds of the data, and it is reasonable to differentiate the data by weighing more/less each one according to its accuracy/reliability. This policy can be implemented by the weighted least-squares (WLS) solution

$$\boldsymbol{\theta}_W^o = \begin{bmatrix} \theta_{W1}^o \\ \theta_{W0}^o \end{bmatrix} = [A^T W A]^{-1} A^T W \ \mathbf{y} \tag{3.8.5}$$

which minimizes the weighted objective function

$$J_W = [A\boldsymbol{\theta} - \mathbf{y}]^T W [A\boldsymbol{\theta} - \mathbf{y}] \tag{3.8.6}$$

If the weighting matrix is $W = V^{-1} = R^{-T} R^{-1}$, then we can write the WLS solution (3.8.5) as

$$\boldsymbol{\theta}_W^o = \begin{bmatrix} \theta_{W1}^o \\ \theta_{W0}^o \end{bmatrix} = [(R^{-1}A)^T (R^{-1}A)]^{-1} (R^{-1}A)^T R^{-1}\mathbf{y} = [A_R^T A_R]^{-1} A_R^T \mathbf{y}_R \tag{3.8.7}$$

where

$$A_R = R^{-1}A, \qquad \mathbf{y}_R = R^{-1}\mathbf{y}, \qquad W = V^{-1} = R^{-T} R^{-1} \tag{3.8.8}$$

One may use the MATLAB built-in routine "`lscov(A,y,V)`" to obtain this WLS solution.

3.8.2 Polynomial Curve Fit: A Polynomial Function of Higher Degree

If there is no reason to limit the degree of fitting polynomial to one, then we may increase the degree of fitting polynomial to, say, N in expectation of decreasing the error. Still, we can use Eq. (3.8.4) or (3.8.6), but with different definitions of A and $\boldsymbol{\theta}$ as

$$A = \begin{bmatrix} x_1^N & \cdot & x_1 & 1 \\ x_2^N & \cdot & x_2 & 1 \\ \cdot & \cdot & \cdot & \cdot \\ x_M^N & \cdot & x_M & 1 \end{bmatrix}, \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_N \\ \cdot \\ \theta_1 \\ \theta_0 \end{bmatrix} \tag{3.8.9}$$

The MATLAB routine "`polyfits()`" performs the WLS or LS scheme to find the coefficients of a polynomial fitting a given set of data points, depending on whether or not a vector (`r`) having the diagonal elements of the weighting matrix W is given as the fourth or fifth input argument. Note that in the case of a diagonal weighting matrix W, the WLS solution conforms to the LS solution with each row of the information matrix A and the data vector $\mathbf{y}$ multiplied by the corresponding element of the weighting matrix W. Let us see the following examples for its usage:

```
function [th,err,yi] = polyfits(x,y,N,xi,r)
%x,y : the row vectors of data pairs
%N   : the order of polynomial(>=0)
%r   : reverse weighting factor array of the same dimension as y
M = length(x); x = x(:); y = y(:); %Make all column vectors
if nargin == 4
  if length(xi) == M, r = xi; xi = x; %With input argument (x,y,N,r)
   else r = 1; %With input argument (x,y,N,xi)
  end
 elseif nargin == 3, xi = x; r = 1;  % With input argument (x,y,N)
end
A(:,N + 1) = ones(M,1);
for n = N:-1:1, A(:,n) = A(:,n+1).*x; end %Eq.(3.8.9)
if length(r) == M
  for m = 1:M, A(m,:) = A(m,:)/r(m); y(m) = y(m)/r(m); end %Eq.(3.8.8)
end
th = (A\y)' %Eq.(3.8.3) or (3.8.7)
ye = polyval(th,x); err = norm(y - ye)/norm(y); %estimated y values, error
yi = polyval(th,xi);
```

```
%do_polyfit
load xy1.dat
x = xy1(:,1); y = xy1(:,2);
[x,i] = sort(x); y = y(i); %sort the data for plotting
xi = min(x)+[0:100]/100*(max(x) - min(x)); %intermediate points
for i = 1:4
  [th,err,yi] = polyfits(x,y,2*i - 1,xi); err %LS
  subplot(220+i)
  plot(x,y,'k*',xi,yi,'b:')
end
```

```
%xy1.dat
-3.0  -0.2774
-2.0   0.8958
-1.0  -1.5651
 0.0   3.4565
 1.0   3.0601
 2.0   4.8568
 3.0   3.8982
```

Example 3.6. Polynomial Curve Fit by LS (Least Squares). Suppose we have an ASCII data file "`xy1.dat`" containing a set of data pairs $\{(x_k, y_k), k = 0{:}6\}$ in two columns and we must fit these data into polynomials of degree 1, 3, 5, and 7.

x	-3	-2	-1	0	1	2	3
y	-0.2774	0.8958	-1.5651	3.4565	3.0601	4.8568	3.8982

We make the MATLAB program "`do_polyfit.m`", which uses the routine "`polyfits()`" to do this job and plot the results together with the given data

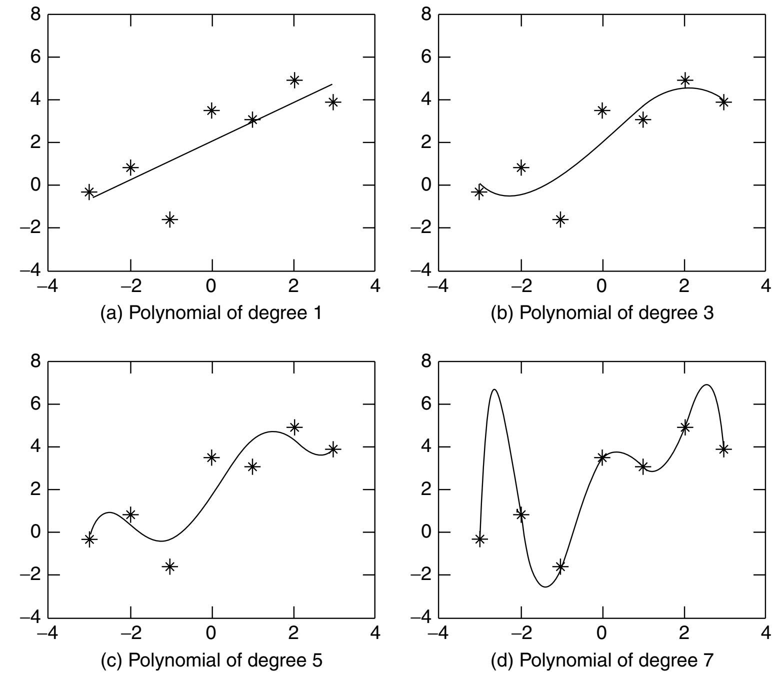


Figure 3.10 Polynomial curve fitting by the LS (Least-Squares) method.

points as depicted in Fig. 3.10. We can observe the polynomial wiggle that the oscillation of the fitting curve between the data points becomes more pronounced with higher degree.

Example 3.7. Curve Fitting by WLS (Weighted Least Squares). Most experimental data have some absolute and/or relative error bounds that are not uniform for all data. If we know the error bounds for each data, we may give each data a weight inversely proportional to the size of its error bound when extracting valuable information from the data. The WLS solution (3.8.7) enables us to reflect such a weighting strategy on estimating data trends. Consider the following two cases.

(a) Suppose there are two gauges A and B with the same function, but different absolute error bounds ± 0.2 and ± 1.0, respectively. We used them to get the input-output data pair (x_m, y_m) as

$$\{(1, 0.0831), (3, 0.9290), (5, 2.4932), (7, 4.9292), (9, 7.9605)\} \quad \text{from gauge A}$$

$$\{(2, 0.9536), (4, 2.4836), (6, 3.4173), (8, 6.3903), (10, 10.2443)\} \quad \text{from gauge B}$$

Let the fitting function be a second-degree polynomial function

$$y = a_2 x^2 + a_1 x + a_0 \tag{E3.7.1}$$

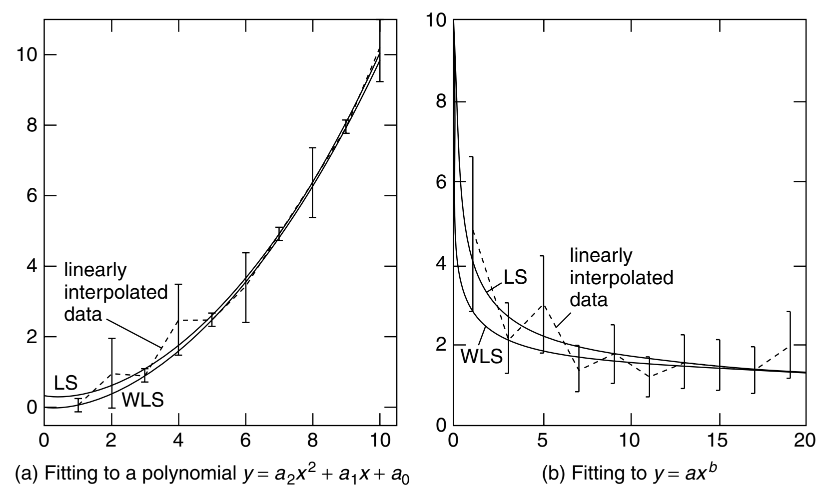


Figure 3.11 LS curve fitting and WLS curve fitting for Example 3.7.

To find the parameters a_2, a_1, and a_0, we write the MATLAB program "`do_wlse1.m`", which uses the routine "`polyfits()`" twice, once without weighting coefficients and once with weighting coefficients. The results are depicted in Fig. 3.11a, which shows that the WLS curve fitting tries to be closer to the data points with smaller error bound, while the LS curve fitting weights all data points equally, which may result in larger deviations from data points with small error bounds.

(b) Suppose we use one gauge that has relative error bound $\pm 40[\%]$ for measuring the output y for the input values $x = [1, 3, 5, \ldots, 19]$ and so the size of error bound of each output data is proportional to the magnitude of the output. We used it to get the input–output data pair (x_m, y_m) as

$$\{(1, 4.7334), (3, 2.1873), (5, 3.0067), (7, 1.4273), (9, 1.7787)$$
$$(11, 1.2301), (13, 1.6052), (15, 1.5353), (17, 1.3985), (19, 2.0211)\}$$

Let the fitting function be an exponential function

$$y = ax^b \tag{E3.7.2}$$

To find the parameters a and b, we make the MATLAB program "`do_wlse2.m`", which uses the routine "`curve_fit()`" without the weighting coefficients one time and with the weighting coefficients another time. The results depicted in Fig. 3.11b shows that the WLS curve fitting tries to get closer to the data points with smaller $|y|$, while the LS curve fitting pays equal respect to all data points, which may result in larger deviation from data points with small $|y|$. Note that the MATLAB routine "`curve_fit()`" appears in Problem 3.11, which implements all of the schemes listed in Table 3.5 with the LS/WLS solution.

(cf) Note that the objective of the WLS scheme is to put greater emphasis on more reliable data.

```
%do_wlse1 for Ex.3.7
clear, clf
x = [1 3 5 7 9 2 4 6 8 10]; %input data
y = [0.0831 0.9290 2.4932 4.9292 7.9605 ...
     0.9536 2.4836 3.4173 6.3903 10.2443]; %output data
eb = [0.2*ones(5,1); ones(5,1)]; %error bound for each y
[x,i] = sort(x); y = y(i); eb = eb(i); %sort the data for plotting
errorbar(x,y,eb,':'), hold on
N = 2; %the degree of the approximate polynomial
xi = [0:100]/10; %interpolation points
[thl,errl,yl] = polyfits(x,y,N,xi);
[thwl,errwl,ywl] = polyfits(x,y,N,xi,eb);
plot(xi,yl,'b', xi,ywl,'r')
%KC = 0; thlc = curve_fit(x,y,KC,N,xi); %for cross-check
%thwlc = curve_fit(x,y,KC,N,xi,eb);
```

```
%do_wlse2
clear, clf
x = [1:2:20]; Nx = length(x); %changing input
xi = [1:200]/10; %interpolation points
eb = 0.4*ones(size(x)); %error bound for each y
y = [4.7334 2.1873 3.0067 1.4273 1.7787 1.2301 1.6052 1.5353 ...
     1.3985 2.0211];
[x,i] = sort(x); y = y(i); eb = eb(i); %sort the data for plotting
eby = y.*eb; %our estimation of error bounds
KC = 6; [thlc,err,yl] = curve_fit(x,y,KC,0,xi);
[thwlc,err,ywl] = curve_fit(x,y,KC,0,xi,eby);
errorbar(x,y,eby), hold on
plot(xi,yl,'b', xi,ywl,'r')
```

3.8.3 Exponential Curve Fit and Other Functions

Why don't we use functions other than the polynomial function as a candidate for fitting functions? There is no reason why we have to stick to the polynomial function, as illustrated in Example 3.7(b). In this section, we consider the case in which the data distribution or the theoretical background behind the data tells us that it is appropriate to fit the data into some nonpolynomial function.

Suppose it is desired to fit the data into the following exponential function.

$$c\ e^{ax} = y \tag{3.8.10}$$

Taking the natural logarithm of both sides, we linearize this as

$$a\ x + \ln c = \ln y \tag{3.8.11}$$

Table 3.5 Linearization of Nonlinear Functions by Parameter/Data Transformation

Function to Fit	Linearized Function	Variable Substitution/ Parameter Restoration
(1) $y = \frac{a}{x} + b$	$y = a\frac{1}{x} + b \to y = ax' + b$	$x' = \frac{1}{x}$
(2) $y = \frac{b}{x+a}$	$\frac{1}{y} = \frac{1}{b}x + \frac{a}{b} \to y' = a'x + b'$	$y' = \frac{1}{y}, a = \frac{b'}{a'}, b = \frac{1}{a'}$
(3) $y = a\ b^x$	$\ln y = (\ln b)x + \ln a$ $\to y' = a'x + b'$	$y' = \ln y, a = e^{b'}, b = e^{a'}$
(4) $y = b\ e^{ax}$	$\ln y = ax + \ln b \to y' = ax + b'$	$y' = \ln y,\ b = e^{b'}$
(5) $y = C - b\ e^{-ax}$	$\ln(C - y) = -ax + \ln b$ $\to y' = a'x + b'$	$y' = \ln(C - y)$ $a = -a', b = e^{b'}$
(6) $y = a\ x^b$	$\ln y = b(\ln x) + \ln a$ $\to y' = a'x' + b'$	$y' = \ln y, x' = \ln x$ $a = e^{b'}, b = a'$
(7) $y = ax\ e^{bx}$	$\ln y - \ln x = bx + \ln a$ $\to y' = a'x + b'$	$y' = \ln(y/x)$ $a = e^{b'}, b = a'$
(8) $y = \frac{C}{1 + b\ e^{ax}}$ $(a\langle 0, b\rangle 0, C = y(\infty))$	$\ln\left(\frac{C}{y} - 1\right) = ax + \ln b$ $\to y' = ax + b'$	$y' = \ln\left(\frac{C}{y} - 1\right), b = e^{b'}$
(9) $y = a \ln x + b$	$\to y = ax' + b$	$x' = \ln x$

so that the LS algorithm (3.8.3) can be applied to estimate the parameters a and $\ln c$ based on the data pairs $\{(x_k, \ln y_k), k = 0 : M\}$.

Like this, there are many other nonlinear relations that can be linearized to fit the LS algorithm, as listed in Table 3.5. This makes us believe in the extensive applicability of the LS algorithm. If you are interested in making a MATLAB routine that implements what are listed in this table, see Problem 3.11, which lets you try the MATLAB built-in function "`lsqcurvefit(f,th0,x,y)`" that enables one to use any type of function (`f`) for curve fitting.

3.9 FOURIER TRANSFORM

Most signals existent in this world contain various frequency components, where rapidly/slowly changing one contains high/low-frequency components. Fourier series/transform is a mathematical tool that can be used to analyze the frequency characteristic of periodic/aperiodic signals. There are four similar definitions of Fourier series/transform, namely, continuous-time Fourier series (CtFS), continuous-time Fourier transform (CtFT), discrete-time Fourier transform (DtFT), and discrete Fourier series/transform (DFS/DFT). Among these tools, DFT can easily and efficiently be programmed in computer languages and that's why we deal with just DFT in this section.

Suppose a sequence of data $\{x[n] = x(nT), n = 0 : M - 1\}(T$: the sampling period) is obtained by sampling a continuous-time/space signal once every T seconds. The $N(\geq M)$-point DFT/IDFT (inverse DFT) pair is defined as

$$\text{DFT:} \quad X(k) = \sum_{n=0}^{N-1} x[n]e^{-j2\pi nk/N}, \qquad k = 0 : N - 1 \tag{3.9.1a}$$

$$\text{IDFT:} \quad x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X(k)e^{j2\pi nk/N}, \qquad n = 0 : N - 1 \tag{3.9.1b}$$

Remark 3.3. DFS/DFT (Discrete Fourier Series/Transform)

0. Note that the indices of the DFT/IDFT sequences appearing in MATLAB range from 1 to N.
1. Generally, the DFT coefficient $X(k)$ is complex-valued and denotes the magnitude and phase of the signal component having the digital frequency $\Omega_k = k\Omega_0 = 2\pi k/N$[rad], which corresponds to the analog frequency $\omega_k = k\omega_0 = k\Omega_0/T = 2\pi k/NT$[rad/s]. We call $\Omega_0 = 2\pi/N$ and $\omega_0 = 2\pi/NT$ (N represents the size of DFT) the digital/analog fundamental or resolution frequency, since it is the minimum digital/analog frequency difference that can be distinguished by the N-point DFT.
2. The DFS and the DFT are essentially the same, but different in the range of time/frequency interval. More specifically, a signal $x[n]$ and its DFT $X(k)$ are of finite duration over the time/frequency range $\{0 \leq n \leq N - 1\}$ and $\{0 \leq k \leq N - 1\}$, respectively, while a signal $\tilde{x}[n]$ (to be analyzed by DFS) and its DFS $\tilde{X}(k)$ are periodic with the period N over the whole set of integers.
3. FFT (fast Fourier transform) means the computationally efficient algorithm developed by exploiting the periodicity and symmetry in the multiplying factor $e^{i2\pi nk/N}$ to reduce the number of complex number multiplications from N^2 to $(N/2)\log_2 N$ (N represents the size of DFT). The MATLAB built-in functions "`fft()`"/"`ifft()`" implement the FFT/IFFT algorithm for the data of length $N = 2^l$ (l represents a nonnegative integer). If the length $M$of the original data sequence is not a power of 2, it can be extended by padding the tail part of the sequence with zeros, which is called zero-padding.

3.9.1 FFT Versus DFT

As mentioned in item 3 of Remark 3.3, FFT/IFFT (inverse FFT) is the computationally efficient algorithm for computing the DFT/IDFT and is fabricated into the MATLAB functions "`fft()`"/"`ifft()`". In order to practice the use of the MATLAB functions and realize the computational advantage of FFT/IFFT over DFT/IDFT, we make the MATLAB program "`compare_dft_fft.m`". Readers are recommended to run this program and compare the execution times consumed by the 1024-point DFT/IDFT computation and its FFT/IFFT scheme, seeing that the

resulting spectra are exactly the same and thus are overlapped onto each other as depicted in Fig. 3.12.

```
%compare_DFT_FFT
clear, clf
N = 2^10; n = [0:N - 1];
x = cos(2*pi*200/N*n)+ 0.5*sin(2*pi*300/N*n);
tic
for k = 0:N - 1, X(k+1) = x*exp(-j*2*pi*k*n/N).'; end %DFT
k = [0:N - 1];
for n = 0:N - 1, xr(n + 1) = X*exp(j*2*pi*k*n/N).'; end %IDFT
time_dft = toc %number of floating-point operations
plot(k,abs(X)), pause, hold on
tic
X1 = fft(x); %FFT
xr1 = ifft(X1); %IFFT
time_fft = toc %number of floating-point operations
clf, plot(k,abs(X1),'r') %magnitude spectrum in Fig. 3.12
```

3.9.2 Physical Meaning of DFT

In order to understand the physical meaning of FFT, we make the MATLAB program "`do_fft`" and run it to get Fig. 3.13, which shows the magnitude spectra of the sampled data taken every T seconds from a two-tone analog signal

$$x(t) = \sin(1.5\pi t) + 0.5\cos(3\pi t) \tag{3.9.2}$$

Readers are recommended to complete the part of this program to get Fig. 3.13c,d and run the program to see the plotting results (see Problem 3.16).

What information do the four spectra for the same analog signal $x(t)$ carry? The magnitude of $X_a(k)$ (Fig. 3.13a) is large at $k = 2$ and 5, each corresponding to $k\omega_0 = 2\pi k/NT = 2\pi k/3.2 = 1.25\pi \approx 1.5\pi$ and $3.125\pi \approx 3\pi$. The magnitude of $X_b(k)$ (Fig. 3.13b) is also large at $k = 2$ and 5, each corresponding to $k\omega_0 = 1.25\pi \approx 1.5\pi$ and $3.125\pi \approx 3\pi$. The magnitude of $X_c(k)$ (Fig. 3.13c) is

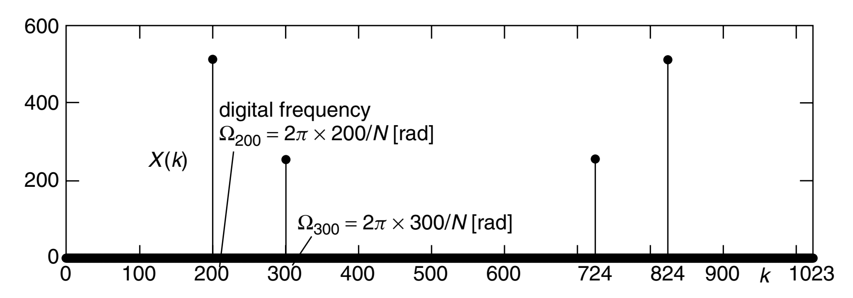


Figure 3.12 The DFT(FFT) $\{X(k), k = 0 : N - 1\}$ of $x[N] = \cos(2\pi \times 200n/N) + 0.5\sin(2\pi \times 300n/N)$ for $n = 0 : N - 1 (N = 2^{10} = 1024)$.

```
%do_fft (to get Fig. 3.13)
clear, clf
w1 = 1.5*pi; w2=3*pi; %two tones
N = 32; n = [0:N - 1]; T = 0.1; %sampling period
t = n*T; xan = sin(w1*t) + 0.5*sin(w2*t);
subplot(421), stem(t,xan,'.')
k = 0:N - 1; Xa = fft(xan);
dscrp=norm(xan-real(ifft(Xa))) %x[n] reconstructible from IFFT{X(k)}?
subplot(423), stem(k,abs(Xa),'.')
%upsampling
N = 64; n = [0:N - 1]; T = 0.05; %sampling period
t = n*T; xbn = sin(w1*t)+ 0.5*sin(w2*t);
subplot(422), stem(t,xbn,'.')
k = 0:N - 1; Xb = fft(xbn);
subplot(424), stem(k,abs(Xb),'.')
%zero-padding
N = 64; n = [0:N-1]; T = 0.1; %sampling period
......................
```

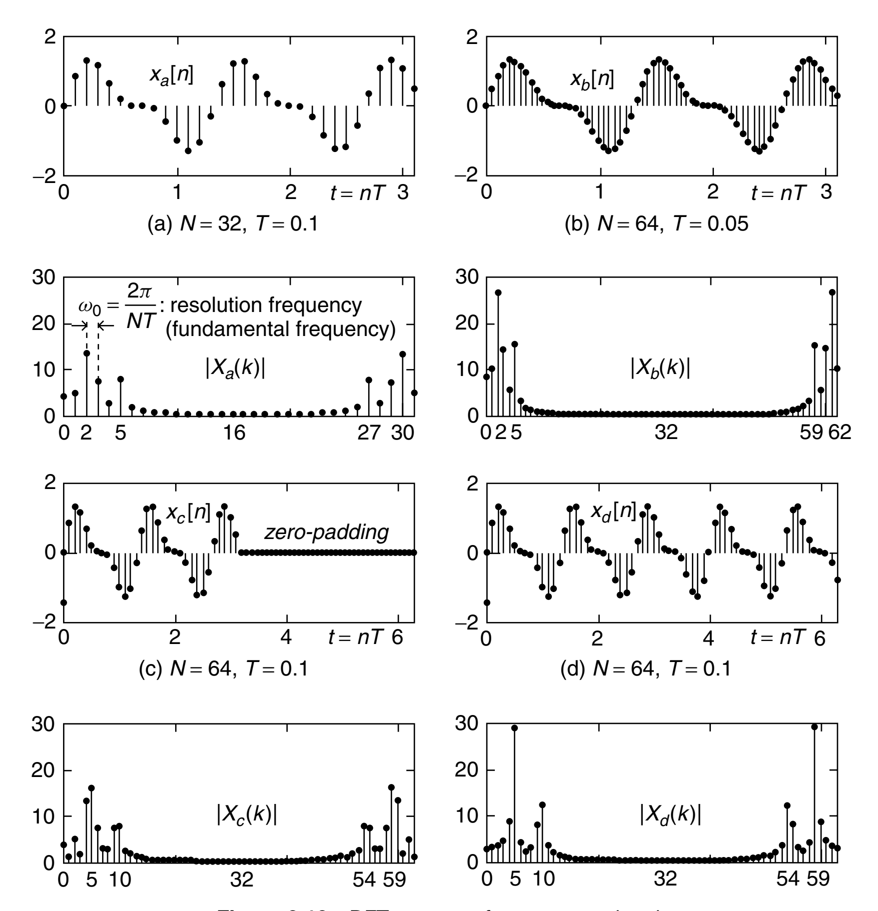


Figure 3.13 DFT spectra of a two-tone signal.

large at $k = 4$,5 and 9,10, and they can be alleged to represent two tones of $k\omega_0 = 2\pi k/NT = 2\pi k/6.4 \approx 1.25\pi \sim 1.5625\pi$ and $2.8125\pi \sim 3.125\pi$. The magnitude of $X_d(k)$ (Fig. 3.13d) is also large at $k = 5$ and 10, each corresponding to $k\omega_0 = 1.5625\ \pi \approx 1.5\ \pi$ and $3.125\ \pi \approx 3\pi$.

It is strange and interesting that we have many different DFT spectra for the same analog signal, depending on the DFT size, the sampling period, the whole interval, and zero-padding. Compared with spectrum (a), spectrum (b) obtained by decreasing the sampling period T from 0.1s to 0.05s has wider analog frequency range $[0,2\pi/T_b]$, but the same analog resolution frequency is $\omega_0 = \Omega_0/T_b = 2\pi/N_bT_b = \pi/1.6 \equiv 2\pi/N_aT_a$; consequently, it does not present us with any new information over (a) for all increased number of data points. The shorter sampling period may be helpful in case the analog signal has some spectral contents of frequency higher than π/T_a. The spectrum (c) obtained by zero-padding has a better-looking, smoother shape, but the vividness is not much improved compared with (a) or (b), since the zeros essentially have no valuable information in the time domain. In contrast with (b) and (c), spectrum (d) obtained by extending the whole time interval shows us the spectral information more distinctly.

Note the following things:

- Zero-padding in the time domain yields the interpolation (smoothing) effect in the frequency domain and vice versa, which will be made use of for data smoothing in the next section (see Problem 3.19).
- If a signal is of finite duration and has the value of zeros outside its domain on the time axis, its spectrum is not discrete, but continuous along the frequency axis, while the spectrum of a periodic signal is discrete as can be seen in Fig. 3.12 or 3.13.
- The DFT values $X(0)$ and $X(N/2)$ represent the spectra of the dc component ($\Omega_0 = 0$) and the virtually highest digital frequency components ($\Omega_{N/2} = N/2 \times 2\pi/N = \pi$ [rad]), respectively.

Here, we have something questionable. The DFT spectrum depicted in Fig. 3.12 shows clearly the digital frequency components $\Omega_{200} = 2\pi \times 200/N$ and $\Omega_{300} = 2\pi \times 300/N$[rad]($N = 2^{10} = 1024$) contained in the discrete-time signal

$$x[n] = \cos(2\pi \times 200n/N) + 0.5\sin(2\pi \times 300n/N), \quad N = 2^{10} = 1024 \tag{3.9.3}$$

and so we can find the analog frequency components $\omega_k = \Omega_k/T$ as long as the sampling period T is known, while the DFT spectra depicted in Fig. 3.13 are so unclear that we cannot discern even the prominent frequency contents. What's wrong with these spectra? It is never a 'right-or-wrong' problem. The only difference is that the digital frequencies contained in the discrete-time signal described by Eq. (3.9.3) are multiples of the fundamental frequency $\Omega_0 = 2\pi/N$, but the analog frequencies contained in the continuous-time signal described by Eq. (3.9.2) are not multiples of the fundamental frequency $\omega_0 = 2\pi/NT$; in other words, the whole time interval $[0, NT)$ is not a multiple of the period of each frequency to be detected. The phenomenon whereby the spectrum becomes

blurred like this is said to be the 'leakage problem'. The leakage problem occurs in most cases because we cannot determine the length of the whole time interval in such a way that it is a multiple of the period of the signal as long as we don't know in advance the frequency contents of the signal. If we knew the frequency contents of a signal, why do we bother to find its spectrum that is already known? As a measure to alleviate the leakage problem, there is a windowing technique [O-1, Section 11.2]. Interested readers can see Problem 3.18.

Also note that the periodicity with period N(the DFT size) of the DFT sequence $X(k)$ as well as $x[n]$, as can be manifested by substituting $k + mN$ (m represents any integer) for k in Eq. (3.9.1a) and also substituting $n + mN$ for n in Eq. (3.9.1b). A real-world example reminding us of the periodicity of DFT spectrum is the so-called stroboscopic effect whereby the wheel of a carriage driven by a horse in the scene of a western movie looks like spinning at lower speed than its real speed or even in the reverse direction. The periodicity of $x[n]$ is surprising, because we cannot imagine that every discrete-time signal is periodic with the period of N, which is the variable size of the DFT to be determined by us. As a matter of fact, the 'weird' periodicity of $x[n]$ can be regarded as a kind of cost that we have to pay for computing the sampled DFT spectrum instead of the continuous spectrum $X(\omega)$ for a continuous-time signal $x(t)$, which is originally defined as

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}\, dt \tag{3.9.4}$$

Actually, this is to blame for the blurred spectra of the two-tone signal depicted in Fig. 3.13.

3.9.3 Interpolation by Using DFS

```
function [xi,Xi] = interpolation_by_DFS(T,x,Ws,ti)
%T : sampling interval (sample period)
%x : discrete-time sequence
%Ws: normalized stop frequency (1.0=pi[rad])
%ti: interpolation time range or # of divisions for T
if nargin < 4, ti = 5; end
if nargin < 3 | Ws > 1, Ws = 1; end
N = length(x);
if length(ti) == 1
  ti = 0:T/ti:(N-1)*T; %subinterval divided by ti
end
ks = ceil(Ws*N/2);
Xi = fft(x);
Xi(ks + 2:N - ks) = zeros(1,N - 2*ks - 1); %filtered spectrum
xi = zeros(1,length(ti));
for k = 2:N/2
   xi = xi+Xi(k)*exp(j*2*pi*(k - 1)*ti/N/T);
end
xi = real(2*xi+Xi(1)+Xi(N/2+1)*cos(pi*ti/T))/N; %Eq.(3.9.5)
```

```
%interpolate_by_DFS
clear, clf
w1 = pi; w2 = .5*pi; %two tones
N = 32; n = [0:N - 1]; T = 0.1; t = n*T;
x = sin(w1*t)+0.5*sin(w2*t)+(rand(1,N) - 0.5); %0.2*sin(20*t);
ti = [0:T/5:(N - 1)*T];
subplot(411), plot(t,x,'k.') %original data sequence
title('original sequence and interpolated signal')
[xi,Xi] = interpolation_by_DFS(T,x,1,ti);
hold on, plot(ti,xi,'r') %reconstructed signal
k = [0:N - 1];
subplot(412), stem(k,abs(Xi),'k.') %original spectrum
title('original spectrum')
[xi,Xi] = interpolation_by_DFS(T,x,1/2,ti);
subplot(413), stem(k,abs(Xi),'r.') %filtered spectrum
title('filtered spectrum')
subplot(414), plot(t,x,'k.', ti,xi,'r') %filtered signal
title('filtered/smoothed signal')
```

We can use the DFS/DFT to interpolate a given sequence $x[n]$ that is supposed to have been obtained by sampling some signal at equidistant points (instants). The procedure consists of two steps; to take the N-point FFT $X(k)$ of $x[n]$ and to use the formula

$$\hat{x}(t) = \frac{1}{N}\sum_{|k|<N/2} \widetilde{X}(k)e^{j2\pi kt/NT}$$

$$= \frac{1}{N}\{X(0) + 2\sum_{k=1}^{N/2-1} \text{Real}\{X(k)e^{j2\pi kt/NT}\} + X(N/2)\cos(\pi t/T)\} \quad (3.9.5)$$

This formula is cast into the routine "`interpolation_by_dfs`", which makes it possible to filter out the high-frequency portion over (`Ws`$\cdot\pi$`,(2-Ws)`π) with `Ws` given as the third input argument. The horizontal (time) range over which you want to interpolate the sequence can be given as the fourth input argument `ti`. We make the MATLAB program "`interpolate_by_dfs`", which applies the routine to interpolate a set of data obtained by sampling at equidistant points along the spatial or temporal axis and run it to get Fig. 3.14. Figure 3.14a shows a data sequence $x[n]$ of length $N = 32$ and its interpolation (reconstruction) $x(t)$ from the 32-point DFS/DFT $X(k)$ (Fig. 3.14b), while Figs. 3.14c and 3.14d show the (zero-padded) DFT spectrum $X'(k)$ with the digital frequency contents higher than $\pi/2$[rad]$(N/4 < k < 3N/4)$ removed and a smoothed interpolation (fitting curve) $x'(t)$ obtained from $X'(k)$, respectively. This can be viewed as the smoothing effect in the time domain by zero-padding in the frequency domain, in duality with the smoothing effect in the frequency domain by zero-padding in the time domain, which was observed in Fig. 3.13c.

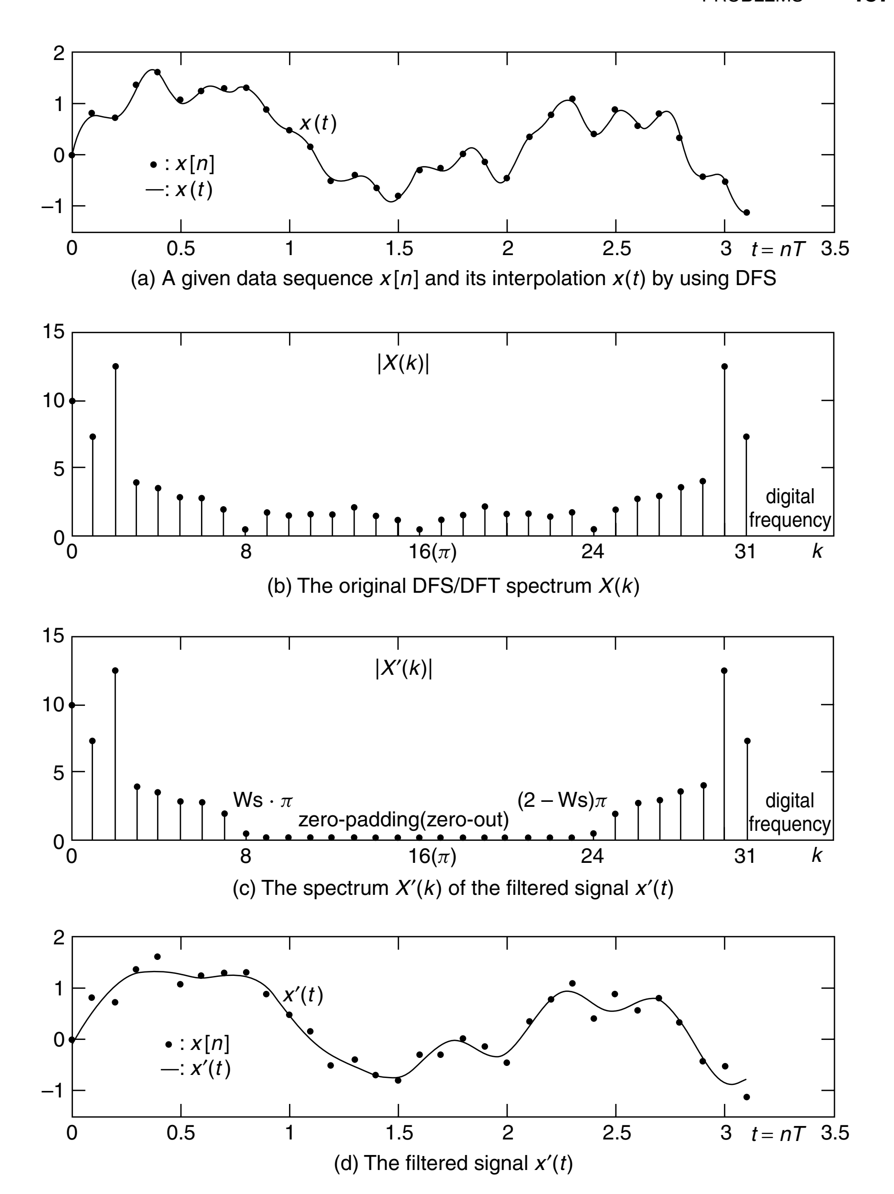


Figure 3.14 Interpolation/smoothing by using DFS/DFT.

PROBLEMS

3.1 Quadratic Interpolation: Lagrange Polynomial and Newton Polynomial

(a) The second-degree Lagrange polynomial matching the three points (x_0, f_0), (x_1, f_1), and (x_2, f_2) can be written by substituting $N = 2$

into Eq. (3.1.3) as

$$l_2(x) = \sum_{m=0}^{2} f_m L_{2,m}(x) = \sum_{m=0}^{2} f_m \prod_{k\neq m}^{N} \frac{x - x_k}{x_m - x_k} \tag{P3.1.1}$$

Check if the zero of the derivative of this polynomial—that is, the root of the equation $l_2'(x) = 0$—is found as

$$\begin{aligned}
l_2'(x) &= f_0 \frac{(x - x_1) + (x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + f_1 \frac{(x - x_2) + (x - x_0)}{(x_1 - x_2)(x_1 - x_0)} \\
&\quad + f_2 \frac{(x - x_0) + (x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \\
&f_0(2x - x_1 - x_2)(x_2 - x_1) + f_1(2x - x_2 - x_0)(x_0 - x_2) \\
&\quad + f_2(2x - x_0 - x_1)(x_1 - x_0) = 0 \\
x = x_3 &= \frac{f_0(x_1^2 - x_2^2) + f_1(x_2^2 - x_0^2) + f_2(x_0^2 - x_1^2)}{2\{f_0(x_1 - x_2) + f_1(x_2 - x_0) + f_2(x_0 - x_1)\}}
\end{aligned} \tag{P3.1.2}$$

You can use the symbolic computation capability of MATLAB by typing the following statements into the MATLAB command window:

```
>>syms x x1 x2 x3 f0 f1 f2
>>L2 = f0*(x - x1)*(x - x2)/(x0 - x1)/(x0 - x2)+...
       f1*(x - x2)*(x - x0)/(x1 - x2)/(x1 - x0)+...
       f2*(x - x0)*(x - x1)/(x2 - x0)/(x2 - x1)
>>pretty(solve(diff(L2)))
```

(b) The second-degree Newton polynomial matching the three points (x_0, f_0), (x_1, f_1), and (x_2, f_2) is Eq. (3.2.4).

$$n_2(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) \tag{P3.1.3}$$

where

$$a_0 = f_0, \qquad a_1 = Df_0 = \frac{f_1 - f_0}{x_1 - x_0}$$

$$a_2 = D^2 f_0 = \frac{Df_1 - Df_0}{x_2 - x_0} = \frac{\dfrac{f_2 - f_1}{x_2 - x_1} - \dfrac{f_1 - f_0}{x_1 - x_0}}{x_2 - x_0} \tag{P3.1.4}$$

Find the zero of the derivative of this polynomial.

(c) From Eq. (P3.1.1) with $x_0 = -1$, $x_1 = 0$, and $x_2 = 1$, find the coefficients of Lagrange coefficient polynomials $L_{2,0}(x)$, $L_{2,1}(x)$, and $L_{2,2}(x)$. You had better make use of the routine "`lagranp()`" for this job.

(d) From the third-degree Lagrange polynomial matching the four points (x_0, f_0), (x_1, f_1), (x_2, f_2), and (x_3, f_3) with $x_0 = -3, x_1 = -2, x_2 = -1$, and $x_3 = 0$, find the coefficients of Lagrange coefficient polynomials $L_{3,0}(x)$, $L_{3,1}(x)$, $L_{3,2}(x)$, and $L_{3,3}(x)$. You had better make use of the routine "`lagranp()`" for this job.

3.2 Error Analysis of Interpolation Polynomial

Consider the error between a true (unknown) function $f(x)$ and the interpolation polynomial $P_N(x)$ of degree N for some $(N + 1)$ points of $y = f(x)$, that is,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_N, y_N)\}$$

where $f(x)$ is up to $(N + 1)$th-order differentiable. Noting that the error is also a function of x and becomes zero at the $(N + 1)$ points, we can write it as

$$e(x) = f(x) - P_N(x) = (x - x_0)(x - x_1) \cdots (x - x_N)g(x) \tag{P3.2.1}$$

Technically, we define an auxiliary function $w(t)$ with respect to t as

$$w(t) = f(t) - P_N(t) - (t - x_0)(t - x_1) \cdots (t - x_N)g(x) \tag{P3.2.2}$$

Then, this function has the value of zero at the $(N + 2)$ points $t = x_0, x_1, \ldots, x_N, x$ and the $1/2/\cdots/(N + 1)$th-order derivative has $(N + 1)/N/\cdots/1$ zeros, respectively. For $t = t_0$ such that $w^{(N+1)}(t_0) = 0$, we have

$$w^{(N+1)}(t_0) = f^{(N+1)}(t_0) - 0 - (N + 1)!g(x) = 0;$$
$$g(x) = \frac{1}{(N + 1)!} f^{(N+1)}(t_0) \tag{P3.2.3}$$

Based on this, show that the error function can be rewritten as

$$e(x) = f(x) - P_N(x) = (x - x_0)(x - x_1) \cdots (x - x_N)\frac{1}{(N + 1)!} f^{(N+1)}(t_0) \tag{P3.2.4}$$

3.3 The Approximation of a Cosine Function

In the way suggested below, find an approximate polynomial of degree 4 for

$$y = f(x) = \cos x \tag{P3.3.1}$$

(a) Find the Lagrange/Newton polynomial of degree 4 matching the following five points and plot the resulting polynomial together with the true function $\cos x$ over $[-\pi, +\pi]$.

k	0	1	2	3	4
x_k	$-\pi$	$-\pi/2$	0	$+\pi/2$	$+\pi$
$f(x_k)$	-1	0	1	0	-1

(b) Find the Lagrange/Newton polynomial of degree 4 matching the following five points and plot the resulting polynomial on the same graph that has the result of (a).

k	0	1	2	3	4
x_k	$\pi\cos(9\pi/10)$	$\pi\cos(7\pi/10)$	0	$\pi\cos(3\pi/10)$	$\pi\cos(\pi/10)$
$f(x_k)$	-0.9882	-0.2723	1	-0.2723	-0.9882

(c) Find the Chebyshev polynomial of degree 4 for $\cos x$ over $[-\pi, +\pi]$ and plot the resulting polynomial on the same graph that has the result of (a) and (b).

3.4 Chebyshev Nodes

The current speed/pressure of the liquid flowing in the pipe, which has irregular radius, will be different from place to place. If you are to install seven speed/pressure gauges through the pipe of length 4 m as depicted in Fig. P3.4, how would you determine the positions of the gauges so that the maximum error of estimating the speed/pressure over the interval [0, 4] can be minimized?

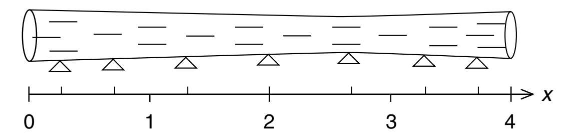


Figure P3.4 Chebyshev nodes.

3.5 Pade Approximation

For the Laplace transform

$$F(s) = e^{-sT} \tag{P3.5.1}$$

representing the delay of T[seconds], we can write its Maclaurin series expansion up to fifth order as

$$Mc(s) \cong 1 - sT + \frac{(sT)^2}{2!} - \frac{(sT)^3}{3!} + \frac{(sT)^4}{4!} - \frac{(sT)^5}{5!} \tag{P3.5.2}$$

(a) Show that we can solve Eq. (3.4.4) and use Eq. (3.4.1) to get the Pade approximation as

$$F(s) \cong p_{1,1}(s) = \frac{q_0 + q_1 s}{1 + d_1 s} = \frac{1 - (T/2)s}{1 + (T/2)s} \cong e^{-Ts} \tag{P3.5.3}$$

(b) Compose a MATLAB program "`nm3p05.m`" that uses the routine "`padeap()`" to generate the Pade approximation of (P3.5.1) with $T = 0.2$ and plots it together with the second-order Maclaurin series expansion and the true function (P3.5.1) for $s = [-5, 10]$. You also run it to see the result as

$$p_{1,1}(s) = \frac{1 - (T/2)s}{1 + (T/2)s} = \frac{-s + 10}{s + 10} \tag{P3.5.4}$$

3.6 Rational Function Interpolation: Bulirsch–Stoer Method [S-3]

Table P3.6 shows the Bulirsch–Stoer method, where its element in the mth row and the $(i + 1)$th column is computed by the following formula:

$$R_m^{i+1} = R_{m+1}^i + \frac{(x - x_{m+i})(R_{m+1}^i - R_{m+1}^{i-1})(R_{m+1}^i - R_m^i)}{(x - x_m)(R_m^i - R_{m+1}^{i-1}) - (x - x_{m+i})(R_{m+1}^i - R_{m+1}^{i-1})}$$

$$\text{with} \quad R_m^o = 0 \text{ and } R_m^1 = y_m \text{ for } i = 1 : N \text{ and } m = 1 : N - i \tag{P3.6.1}$$

```
function yi = rational_interpolation(x,y,xi)
N = length(x); Ni = length(xi);
R(:,1) = y(:);
for n = 1:Ni
  xn = xi(n);
  for i = 1:N - 1
     for m = 1:N - i
        RR1 = R(m + 1,i); RR2 = R(m,i);
        if i > 1,
          RR1 = RR1 - R(m + 1,???); RR2 = RR2 - R(???,i - 1);
        end
        tmp1 = (xn-x(???))*RR1;
        num = tmp1*(R(???,i) - R(m,?));
        den = (xn - x(?))*RR2 -tmp1;
        R(m,i + 1) = R(m + 1,i) ????????;
     end
  end
  yi(n) = R(1,N);
end
```

Table P3.6 Bulirsch–Stoer Method for Rational Function Interpolation

Data	$i = 1$	$i = 2$	$i = 3$	$i = 4$
(x_1, y_1)	$R_1^1 = y_1$	R_1^2	R_1^3	R_1^4
(x_2, y_2)	$R_2^1 = y_2$	R_2^2	R_2^3	.
(x_3, y_3)	$R_3^1 = y_3$	R_3^2	.	
.	.	.		
(x_m, y_m)	.	.		

(a) The above routine "`rational_interpolation(x,y,xi)`" uses the Bulirsch–Stoer method to interpolate the set of data pairs `(x,y)` given as its first/second input arguments over a set of intermediate points `xi` given as its third input argument. Complete the routine and apply it to interpolate the four data points $\{(-1, f(-1)), (-0.2, f(-0.2)), (0.1, f(0.1)), (0.8, f(0.8))\}$ on the graph of $f(x) = 1/(1 + 8x^2)$ for `xi = [-100:100]/100` and plot the interpolated curve together with the graph of the true function $f(x)$. Does it work well? How about doing the same job with another routine "`rat_interp()`" listed in Section 8.3 of [F-1]? What are the values of `yi([95:97])` obtained from the two routines? If you come across anything odd in the graphic results and/or the output numbers, what is your explanation?

(cf) MATLAB expresses the in-determinant 0/0 (zero-divided-by-zero) as NaN (Not-a-Number) and skips the value when plotting it on a graph. It may, therefore, be better off for the plotting purpose if we take no special consideration into the case of in-determinant.

(b) Apply the Pade approximation routine "`padeap()`" (with `M = 2` & `N = 2`) to generate the rational function approximating $f(x) = 1/(1 + 8x^2)$ and compare the result with the true function $f(x)$.

(c) To compare the rational interpolation method with the Pade approximation scheme, apply the routines `rational_interpolation()` and `padeap()` (with `M = 3` & `N = 2`) to interpolate the four data points $\{(-2, f(-2)), (-1, f(-1)), (1, f(1)), (2, f(2))\}$ on the graph of $f(x) = \sin(x)$ for `xi = [-100:100]*pi/100` and plot the interpolated curve together with the graph of the true function. How do you compare the approximation/interpolation results?

3.7 Smoothness of a Cubic Spline Function

We claim that the cubic spline interpolation function $s(x)$ has the smoothness property of

$$\int_{x_k}^{x_{k+1}} (s''(x))^2\,dx \le \int_{x_k}^{x_{k+1}} (f''(x))^2\,dx \tag{P3.7.1}$$

for any second-order differentiable function $f(x)$ matching the given grid points and having the same first-order derivatives as $s(x)$ at the grid points. This implies that the cubic spline functions are not so rugged. Prove it by doing the following.

(a) Check the validity of the equality

$$\int_{x_k}^{x_{k+1}} f''(x)s''(x)\,dx = \int_{x_k}^{x_{k+1}} (s''(x))^2\,dx \tag{P3.7.2}$$

where the left-hand and right-hand sides of this equation are

$$\text{LHS: } \int_{x_k}^{x_{k+1}} f''(x)s''(x)\,dx$$
$$= f'(x)s''(x)|_{x_k}^{x_{k+1}} - \int_{x_k}^{x_{k+1}} f'(x)s'''(x)\,dx$$
$$= f'(x_{k+1})s''(x_{k+1}) - f'(x_k)s''(x_k) - C(f(x_{k+1}) - f(x_k)) \quad \text{(P3.7.3a)}$$
$$\text{RHS: } \int_{x_k}^{x_{k+1}} s''(x)s''(x)\,dx$$
$$= s'(x_{k+1})s''(x_{k+1}) - s'(x_k)s''(x_k) - C(s(x_{k+1}) - s(x_k)) \quad \text{(P3.7.3b)}$$

(b) Check the validity of the following inequality:

$$0 \le \int_{x_k}^{x_{k+1}} (f''(x) - s''(x))^2\,dx$$
$$= \int_{x_k}^{x_{k+1}} (f''(x))^2\,dx - 2\int_{x_k}^{x_{k+1}} f''(x)s''(x)\,dx + \int_{x_k}^{x_{k+1}} (s''(x))^2\,dx$$
$$\overset{\text{(P3.7.2)}}{=} \int_{x_k}^{x_{k+1}} (f''(x))^2\,dx - \int_{x_k}^{x_{k+1}} (s''(x))^2\,dx$$
$$\int_{x_k}^{x_{k+1}} (f''(x))^2\,dx \le \int_{x_k}^{x_{k+1}} (s''(x))^2\,dx \quad \text{(P3.7.4)}$$

3.8 MATLAB Built-in Routine for Cubic Spline

There are two MATLAB built-in routines:

```
>>yi = spline(x,y,xi);
>>yi = interp1(x,y,xi,'spline');
```

Both receive a set of data points `(x,y)` and return the values of the cubic spline interpolating function $s(x)$ for the (intermediate) points `xi` given as the third input argument. Write a program that uses these MATLAB routines to get the interpolation for the set of data points

$$\{(0, 0), (0.5, 2), (2, -2), (3.5, 2), (4, 0)\}$$

and plots the results for [0, 4]. In this program, append the statements that do the same job by using the routine "`cspline(x,y,KC)`" (Section 3.5) with `KC` $= 1, 2,$ and 3. Which one yields the same result as the MATLAB built-in routine? What kind of boundary condition does the MATLAB built-in routine assume?

3.9 Robot Path Planning Using Cubic Spline

Every object having a mass is subject to the law of inertia and so its speed described by the first derivative of its displacement with respect to time must be continuous in any direction. In this context, the cubic spline having the continuous derivatives up to second order presents a good basis for planning the robot path/trajectory. We will determine the path of a robot in such a way that the following conditions are satisfied:

- At time $t = 0$ s, the robot starts from its home position (0, 0) with zero initial velocity, passing through the intermediate point (1, 1) at $t = 1$ s and arriving at the final point (2, 4) at $t = 2$ s.
- On arriving at (2, 4), it starts the point at $t = 2$ s, stopping by the intermediate point (3, 3) at $t = 3$ s and arriving at the point (4, 2) at $t = 4$ s.
- On arriving at (4, 2), it starts the point, passing through the intermediate point (2,1) at $t = 5$ s and then returning to the home position (0, 0) at $t = 6$ s.

More specifically, what we need is

- the spline interpolation matching the three points (0, 0),(1, 1),(2, 2) and having zero velocity at both boundary points (0, 0) and (2, 2),
- the spline interpolation matching the three points (2, 2),(3, 3),(4, 4) and having zero velocity at both boundary points (2, 2) and (4, 4), and
- the spline interpolation matching the three points (4, 4), (5, 2), (6, 0) and having zero velocity at both boundary points (4, 4) and (6, 0) on the tx plane.

On the ty plane, we need

- the spline interpolation matching the three points (0, 0),(1, 1),(2, 4) and having zero velocity at both boundary points (0, 0) and (2, 4),
- the spline interpolation matching the three points (2, 4),(3, 3),(4, 2) and having zero velocity at both boundary points (2, 4) and (4, 2), and
- the spline interpolation matching the three points (4, 2),(5, 1),(6, 0) and having zero velocity at both boundary points (4, 2) and (6, 0).

Supplement the following incomplete program "`robot_path`", whose objective is to make the required spline interpolations and plot the whole robot path obtained through the interpolations on the xy plane. Run it to get the graph as depicted in Fig. P3.9c.

```
%robot_path
x1 = [0  1  2]; y1 = [0  1  4];  t1 = [0  1  2]; ti1 = [0: 0.05: 2];
xi1 = cspline(t1,x1,ti1);  yi1 = cspline(t1,y1,ti1);
.......................................
plot(xi1,yi1,'k', xi2,yi2,'b', xi3,yi3, 'k'), hold on
plot([x1(1) x2(1) x3(1) x3(end)],[y1(1) y2(1) y3(1) y3(end)],'o')
plot([x1 x2 x3],[y1 y2 y3],'k+'), axis([0 5 0 5])
```

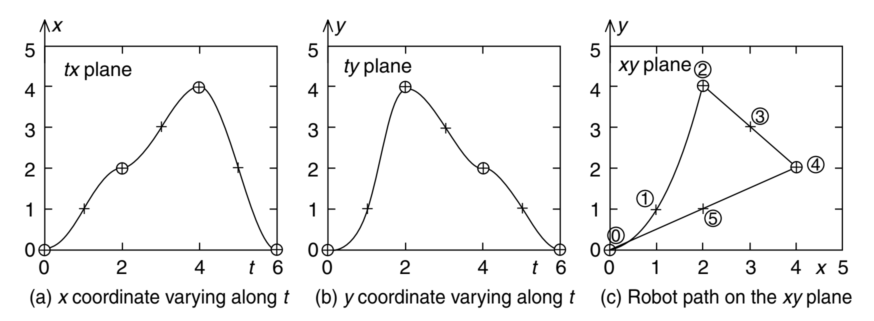


Figure P3.9 Robot path planning using the cubic spline interpolation.

3.10 One-Dimensional Interpolation

What do you have to give as the fourth input argument of the MATLAB built-in routine "`interp1()`" in order to get the same result as that would be obtained by using the following one-dimensional interpolation routine "`intrp1()`"? What letter would you see if you apply this routine to interpolate the data points {(0,3), (1,0), (2,3), (3,0), (4,3)} for [0,4]?

```
function yi = intrp1(x,y,xi)
M = length(x); Mi = length(xi);
for mi = 1: Mi
  if xi(mi) < x(1), yi(mi) = y(1)-(y(2) - y(1))/(x(2) - x(1))*(x(1) - xi(mi));
   elseif xi(mi)>x(M)
     yi(mi) = y(M)+(y(M) - y(M - 1))/(x(M) - x(M-1))*(xi(mi) - x(M));
   else
     for m = 2:M
       if xi(mi) <= x(m)
         yi(mi) = y(m - 1)+(y(m) - y(m - 1))/(x(m) - x(m - 1))*(xi(mi) - x(m - 1));
         break;
       end
     end
  end
end
```

3.11 Least-Squares Curve Fitting

(a) There are several nonlinear relations listed in Table 3.5, which can be linearized to fit the LS algorithm. The MATLAB routine "`curve_fit()`" implements all the schemes that use the LS method to find the parameters for the template relations, but the parts for the relations (1), (2), (7), (8), and (9) are missing. Supplement the missing parts to complete the routine.

(b) The program "`nm3p11.m`" generates the 12 sets of data pairs according to various types of relations (functions), applies the routines "`curve_fit()`"/"`lsqcurvefit()`" to find the parameters of the template relations, and plots the data pairs on the fitting curves obtained from the template functions with the estimated parameters. Complete and run it to get the graphs like Fig. P3.11. Answer the following questions.

(i) If any, find the case(s) where the results of using the two routines make a great difference. For the case(s), try with another initial guess th0 = [1 1] of parameters, instead of th0 = [0 0].

(ii) If the MATLAB built-in routine "`lsqcurvefit()`" yields a bad result, does it always give you a warning message? How do you compare the two routines?

```
function [th,err,yi] = curve_fit(x,y,KC,C,xi,sig)
% implements the various LS curve-fitting schemes in Table 3.5
% KC = the # of scheme in Table 3.5
% C = optional constant (final value) for KC! = 0 (nonlinear LS)
%     degree of approximate polynomial for KC = 0 (standard LS)
% sig = the inverse of weighting factor for WLS
Nx = length(x); x = x(:); y = y(:);
if nargin == 6, sig = sig(:);
  elseif length(xi) == Nx, sig = xi(:); xi = x;
  else sig = ones(Nx,1);
end
if nargin < 5, xi = x; end;  if nargin < 4 | C < 1, C = 1; end
switch KC
  case 1
     ............................
  case 2
     ............................
  case {3,4}
    A(1:Nx,:) = [x./sig ones(Nx,1)./sig];
    RHS = log(y)./sig;    th = A\RHS;
    yi = exp(th(1)*xi + th(2)); y2 = exp(th(1)*x + th(2));
    if KC == 3, th = exp([th(2) th(1)]);
     else  th(2) = exp(th(2));
    end
  case 5
    if nargin < 5, C = max(y) + 1; end %final value
    A(1:Nx,:) = [x./sig ones(Nx,1)./sig];
    y1 = y; y1(find(y > C - 0.01)) = C - 0.01;
    RHS = log(C-y1)./sig;    th = A\RHS;
    yi = C - exp(th(1)*xi + th(2)); y2 = C - exp(th(1)*x + th(2));
    th = [-th(1) exp(th(2))];
  case 6
    A(1:Nx,:) = [log(x)./sig ones(Nx,1)./sig];
    y1 = y; y1(find(y < 0.01)) = 0.01;
    RHS = log(y1)./sig;    th = A\RHS;
    yi = exp(th(1)*log(xi) + th(2)); y2 = exp(th(1)*log(x) + th(2));
    th = [exp(th(2)) th(1)];
  case 7 ............................
  case 8 ............................
  case 9 ............................
  otherwise %standard LS with degree C
    A(1:Nx,C + 1) = ones(Nx,1)./sig;
    for n = C:-1:1, A(1:Nx,n) = A(1:Nx,n + 1).*x;  end
    RHS = y./sig;    th = A\RHS;
    yi = th(C+1);  tmp = ones(size(xi));
    y2 = th(C+1);  tmp2 = ones(size(x));
    for n = C:-1:1,
       tmp = tmp.*xi; yi = yi + th(n)*tmp;
       tmp2 = tmp2.*x; y2 = y2 + th(n)*tmp2;
    end
end
th = th(:)'; err = norm(y - y2);
if nargout == 0, plot(x,y,'*', xi,yi,'k-'); end
```

```
%nm3p11 to plot Fig.P3.11 by curve fitting
clear
x = [1: 20]*2 - 0.1; Nx = length(x);
noise = rand(1,Nx) - 0.5; % 1xNx random noise generator
xi = [1:40]-0.5; %interpolation points
figure(1), clf
a = 0.1; b = -1; c = -50;         %Table 3.5(0)
y = a*x.^2 + b*x + c + 10*noise(1:Nx);
[th,err,yi] = curve_fit(x,y,0,2,xi); [a b c],th
[a b c],th %if you want parameters
f = inline('th(1)*x.^2 + th(2)*x+th(3)','th','x');
[th,err] = lsqcurvefit(f,[0 0 0],x,y), yi1 = f(th,xi);
subplot(321), plot(x,y,'*', xi,yi,'k', xi,yi1,'r')
a = 2; b = 1; y = a./x + b + 0.1*noise(1:Nx);  %Table 3.5(1)
[th,err,yi] = curve_fit(x,y,1,0,xi); [a b],th
f = inline('th(1)./x + th(2)','th','x');
th0 = [0 0]; [th,err] = lsqcurvefit(f,th0,x,y), yi1 = f(th,xi);
subplot(322), plot(x,y,'*', xi,yi,'k', xi,yi1,'r')
a = -20; b = -9; y = b./(x+a) + 0.4*noise(1:Nx); %Table 3.5(2)
[th,err,yi] = curve_fit(x,y,2,0,xi); [a b],th
f = inline('th(2)./(x+th(1))','th','x');
th0 = [0 0]; [th,err] = lsqcurvefit(f,th0,x,y), yi1 = f(th,xi);
subplot(323), plot(x,y,'*', xi,yi,'k', xi,yi1,'r')
a = 2.; b = 0.95; y = a*b.^x + 0.5*noise(1:Nx); %Table 3.5(3)
[th,err,yi] = curve_fit(x,y,3,0,xi); [a b],th
f = inline('th(1)*th(2).^x','th','x');
th0 = [0 0]; [th,err] = lsqcurvefit(f,th0,x,y), yi1 = f(th,xi);
subplot(324), plot(x,y,'*', xi,yi,'k', xi,yi1,'r')
a = 0.1; b = 1; y = b*exp(a*x) +2*noise(1:Nx); %Table 3.5(4)
[th,err,yi] = curve_fit(x,y,4,0,xi); [a b],th
f = inline('th(2)*exp(th(1)*x)','th','x');
th0 = [0 0]; [th,err] = lsqcurvefit(f,th0,x,y), yi1 = f(th,xi);
subplot(325), plot(x,y,'*', xi,yi,'k', xi,yi1,'r')
a = 0.1; b = 1;              %Table 3.5(5)
y = -b*exp(-a*x); C = -min(y)+1; y = C + y + 0.1*noise(1:Nx);
[th,err,yi] = curve_fit(x,y,5,C,xi); [a b],th
f = inline('1-th(2)*exp(-th(1)*x)','th','x');
th0 = [0 0]; [th,err] = lsqcurvefit(f,th0,x,y), yi1 = f(th,xi);
subplot(326), plot(x,y,'*', xi,yi,'k', xi,yi1,'r')
figure(2), clf
a = 0.5; b = 0.5;  y = a*x.^b +0.2*noise(1:Nx); %Table 3.5(6a)
[th,err,yi] = curve_fit(x,y,0,2,xi); [a b],th
f = inline('th(1)*x.^th(2)','th','x');
th0 = [0 0]; [th,err] = lsqcurvefit(f,th0,x,y), yi1 = f(th,xi);
subplot(321), plot(x,y,'*', xi,yi,'k', xi,yi1,'r')
a = 0.5; b = -0.5;              %Table 3.5(6b)
y = a*x.^b + 0.05*noise(1:Nx);
[th,err,yi] = curve_fit(x,y,6,0,xi); [a b],th
f = inline('th(1)*x.^th(2)','th','x');
th0 = [0 0]; [th,err] = lsqcurvefit(f,th0,x,y), yi1 = f(th,xi);
subplot(322), plot(x,y,'*', xi,yi,'k', xi,yi1,'r')
```

(cf) If there is no theoretical basis on which we can infer the physical relation between the variables, how do we determine the candidate function suitable for fitting the data pairs? We can plot the graph of data pairs and choose one of the graphs in Fig. P3.11 which is closest to it and choose the corresponding template function as the candidate fitting function.

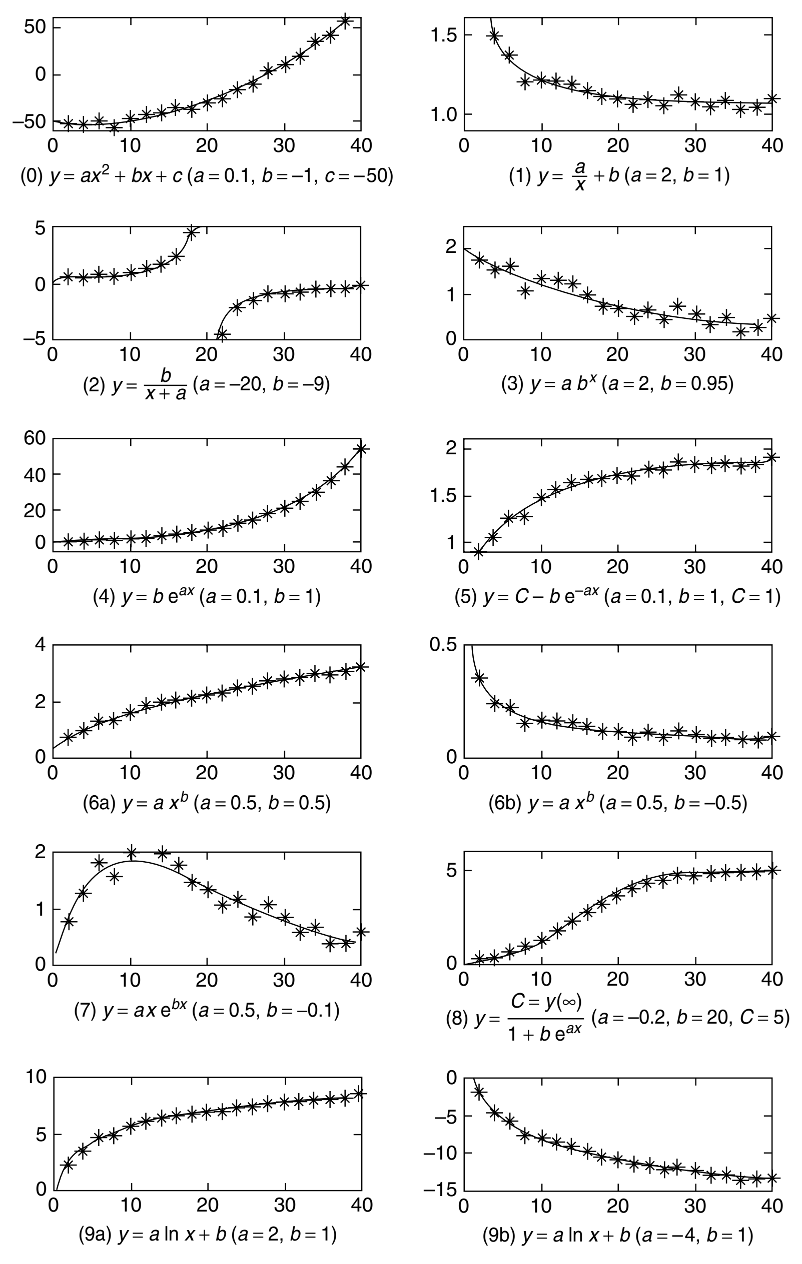


Figure P3.11 LS fitting curves for data pairs with various relations.

3.12 Two-Dimensional Interpolation

Compose a routine "`z = find_depth(xi,yi)`" that finds the depth `z` of a geological stratum at a point `(xi,yi)` given as the input arguments, based on the data in Problem 1.4.

(cf) If you have no idea, insert just one statement involving '`interp2()`' into the program '`nm1p04.m`' (Problem 1.4) and fit it into the format of a MATLAB function.

3.13 Polynomial Curve Fitting by Least Squares and Persistent Excitation

Suppose the theoretical (true) relationship between the input x and the output y is known as

$$y = x + 2 \tag{P3.13.1}$$

Charley measured the output data y 10 times for the same input value $x = 1$ by using a gauge whose measurement errors has a uniform distribution U[−0.5, +0.5]. He made the following MATLAB program "`nm3p13`", which uses the routine "`polyfits()`" to find a straight line fitting the data.

(a) Check the following program and modify it if needed. Then, run the program and see the result. Isn't it beyond your imagination? If you use the MATLAB built-in function "`polyfit()`", does it get any better?

```
%nm3p13.m
tho = [1 2]; %true parameter
x = ones(1,10); %the unchanged input
y = tho(1)*x + tho(2)+(rand(size(x)) - 0.5);
th_ls = polyfits(x,y,1); %uses the MATLAB routine in Sec.3.8.2
polyfit(x,y,1) %uses MATLAB built-in function
```

(b) Note that substituting Eq. (3.8.2) into Eq.(3.8.3) yields

$$\boldsymbol{\theta}^{\mathrm{o}} = \begin{bmatrix} a^{\mathrm{o}} \\ b^{\mathrm{o}} \end{bmatrix} = [A^T A]^{-1} A^T \mathbf{y}$$

$$= \begin{bmatrix} \sum_{n=0}^{M} x_n^2 & \sum_{n=0}^{M} x_n \\ \sum_{n=0}^{M} x_n & \sum_{n=0}^{M} 1 \end{bmatrix}^{-1} \begin{bmatrix} \sum_{n=0}^{M} x_n y_n \\ \sum_{n=0}^{M} y_n \end{bmatrix} \tag{P3.13.2}$$

If $x_n = c(\text{constant})\ \forall\, n = 0 : M$, is the matrix $A^T A$ invertible?

(c) What conclusion can you derive based on (a) and (b), with reference to the identifiability condition that the input must be rich in some sense or persistently exciting?

(cf) This problem implies that the performance of the identification/estimation scheme including the curve fitting depends on the characteristic of input as well as the choice of algorithm.

3.14 Scaled Curve Fitting for an Ill-Conditioned Problem [M-2]

Consider Eq. (P3.13.2), which is a typical least-squares (LS) solution. The matrix $A^T A$, which must be inverted for the solution to be obtained, may become ill-conditioned by the widely different orders of magnitude of its elements, if the magnitudes of all x_n's are too large or too small, being far

from 1 (see Remark 2.3). You will realize something about this issue after solving this problem.

(a) Find a polynomial of degree 2 which fits four data points $(10^6, 1)$, $(1.1 \times 10^6, 2)$, $(1.2 \times 10^6, 5)$, and $(1.3 \times 10^6, 10)$ and plot the polynomial function (together with the data points) over the interval $[10^6, 1.3 \times 10^6]$ to check whether it fits the data points well. How big is the relative mismatch error? Does the polynomial do the fitting job well?

(b) Find a polynomial of degree 2 which fits four data points $(10^7, 1)$,$(1.1 \times 10^7, 2)$, $(1.2 \times 10^7, 5)$, and $(1.3 \times 10^7, 10)$ and plot the polynomial function (together with the data points) over the interval $[10^7, 1.3 \times 10^7]$ to check whether it fits the data points well. How big is the relative mismatch error? Does the polynomial do the fitting job well? Did you get any warning message on the MATLAB command window? What do you think about it?

(c) If you are not satisfied with the result obtained in (b), why don't you try the scaled curve fitting scheme described below?

1. Transform the x_n's of the data point (x_n, y_n)'s into the region $[-2, 2]$ by the following relation.

$$x'_n \leftarrow -2 + \frac{4}{x_{max} - x_{min}}(x_n - x_{min}) \tag{P3.14.1}$$

2. Find the LS polynomial $p(x')$ fitting the data point (x'_n, y_n)'s.

3. Substitute

$$x' \leftarrow -2 + \frac{4}{x_{max} - x_{min}}(x - x_{min}) \tag{P3.14.2}$$

for x' into $p(x')$.

(cf) You can complete the following program "nm3p14" and run it to get the numeric answers.

```
%nm3p14.m
clear, clf
format long e
x = 1e6*[1 1.1 1.2 1.3]; y = [1 2 5 10];
xi = x(1)+[0:1000]/1000*(x(end) - x(1));
[p,err,yi] = curve_fit(x,y,0,2,xi); p, err
plot(x,y,'o',xi,yi), hold on
xmin = min(x); xmax = max(x);
x1 = -2 + 4*(x-xmin)/(xmax - xmin);
x1i = ??????????????????????????;
[p1,err,yi] = ????????????????????????; p1, err
plot(x,y,'o',xi,yi)
%To get the coefficients of the original fitting polynomial
ps1 = poly2sym(p1);
syms x; ps0 = subs(ps1,x, - 2 + 4/(xmax - xmin)*(x - xmin));
p0 = sym2poly(ps0)
format short
```

3.15 Weighted Least-Squares Curve Fitting

As in Example 3.7, we want to compare the results of applying the LS approach and the WLS approach for finding a function that we can believe will describe the relation between the input x and the output y as

$$y = a\ x\ e^{bx} \tag{P3.15}$$

where the data pair $(x_m,\ y_m)$'s are given as

$\{(1, 3.2908), (5, 3.3264), (9, 1.1640), (13, 0.3515), (17, 0.1140)\}$
from gauge A with error range ± 0.1

$\{(3, 4.7323), (7, 2.4149), (11, 0.3814), (15, -0.2396), (19, -0.2615)\}$
from gauge B with error range ± 0.5

Noting that this corresponds to the case of Table 3.5(7), use the MATLAB routine "`curve_fit()`" for this job and get the result as depicted in Fig. P3.15. Identify which one of the two lines a and b is the WLS fitting curve. How do you compare the results?

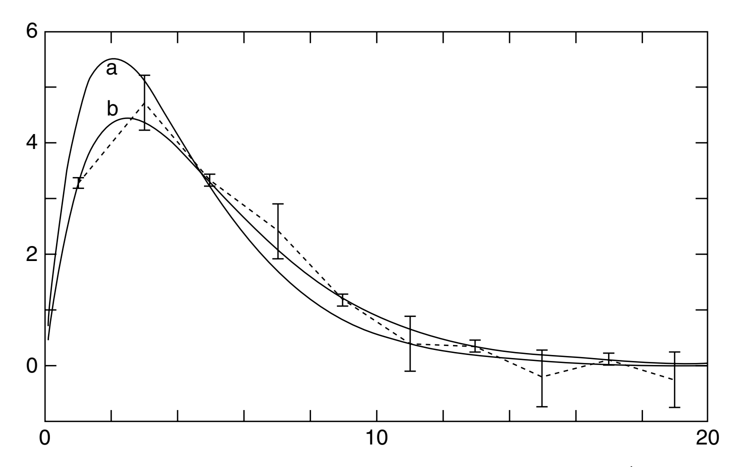


Figure P3.15 The LS and WLS fitting curves to $y = axe^{bx}$.

3.16 DFT (Discrete Fourier Transform) Spectrum

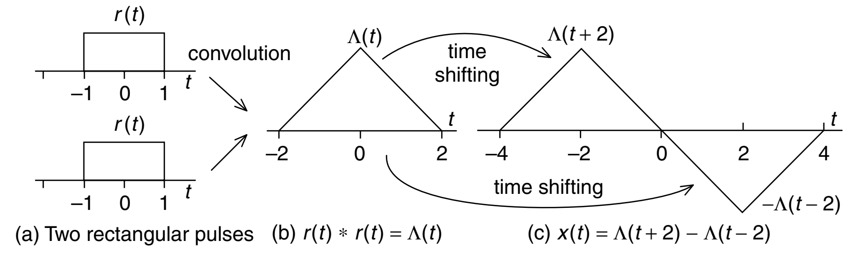
Supplement the part of the MATLAB program "`do_fft`" (Section 3.9.2), which computes the DFT spectra of the two-tone analog signal described by Eq. (3.9.2) for the cases of zero-padding and whole interval extension and plots them as in Figs. 3.13c and 3.13d. Which is the clearest one among the four spectra depicted in Fig. 3.13? If you can generalize this, which would you choose among up-sampling, zero-padding, and whole interval extension to get a clear spectrum?

3.17 Effect of Sampling Period, Zero-Padding, and Whole Time Interval on DFT Spectrum

In Section 3.9.2, we experienced the effect of zero-padding, sampling period reduction, and whole interval extension on the DFT spectrum of a two-tone signal that has two distinct frequency components. Here, we are going to investigate the effect of zero-padding, sampling period reduction, and whole interval extension on the DFT spectrum of a triangular pulse depicted in Fig. P3.17.1c. Additionally, we will compare the DFT with the CtFT (continuous-time Fourier transform) and the DtFT (discrete-time Fourier transform) [O-1].

(a) The definition of CtFT that is used for getting the spectrum of a continuous-time finite-duration signal $x(t)$ is

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}\,dt \tag{P3.17.1}$$



(a) Two rectangular pulses (b) $r(t) * r(t) = \Lambda(t)$ (c) $x(t) = \Lambda(t+2) - \Lambda(t-2)$

Figure P3.17.1 A triangular pulse as the convolution of two rectangular pulses.

The CtFT has several useful properties including the convolution property and the time-shifting property described as

$$x(t) * y(t) \xrightarrow{\text{(CtFT)}} X(\omega)Y(\omega) \tag{P3.17.2}$$

$$x(t - t_1) \xrightarrow{\text{(CtFT)}} X(\omega)e^{-j\omega t_1} \tag{P3.17.3}$$

Noting that the triangular pulse is the convolution of the two rectangular pulse $r(t)$'s whose CtFTs are

$$R(\omega) = \text{CtFT}\{r(t)\} = \int_{-1}^{1} e^{j\omega t}\,dt = 2\frac{\sin\omega}{\omega}$$

we can use the convolution property (P3.17.2) to get the CtFT of the triangular pulse as

$$\text{CtFT}\{\Lambda(t)\} = \text{CtFT}\{r(t) * r(t)\} \overset{\text{(P3.17.2)}}{=} R(\omega)R(\omega)$$

$$= 4\frac{\sin^2\omega}{\omega^2} = 4\sin c^2\left(\frac{\omega}{\pi}\right) \tag{P3.17.4}$$

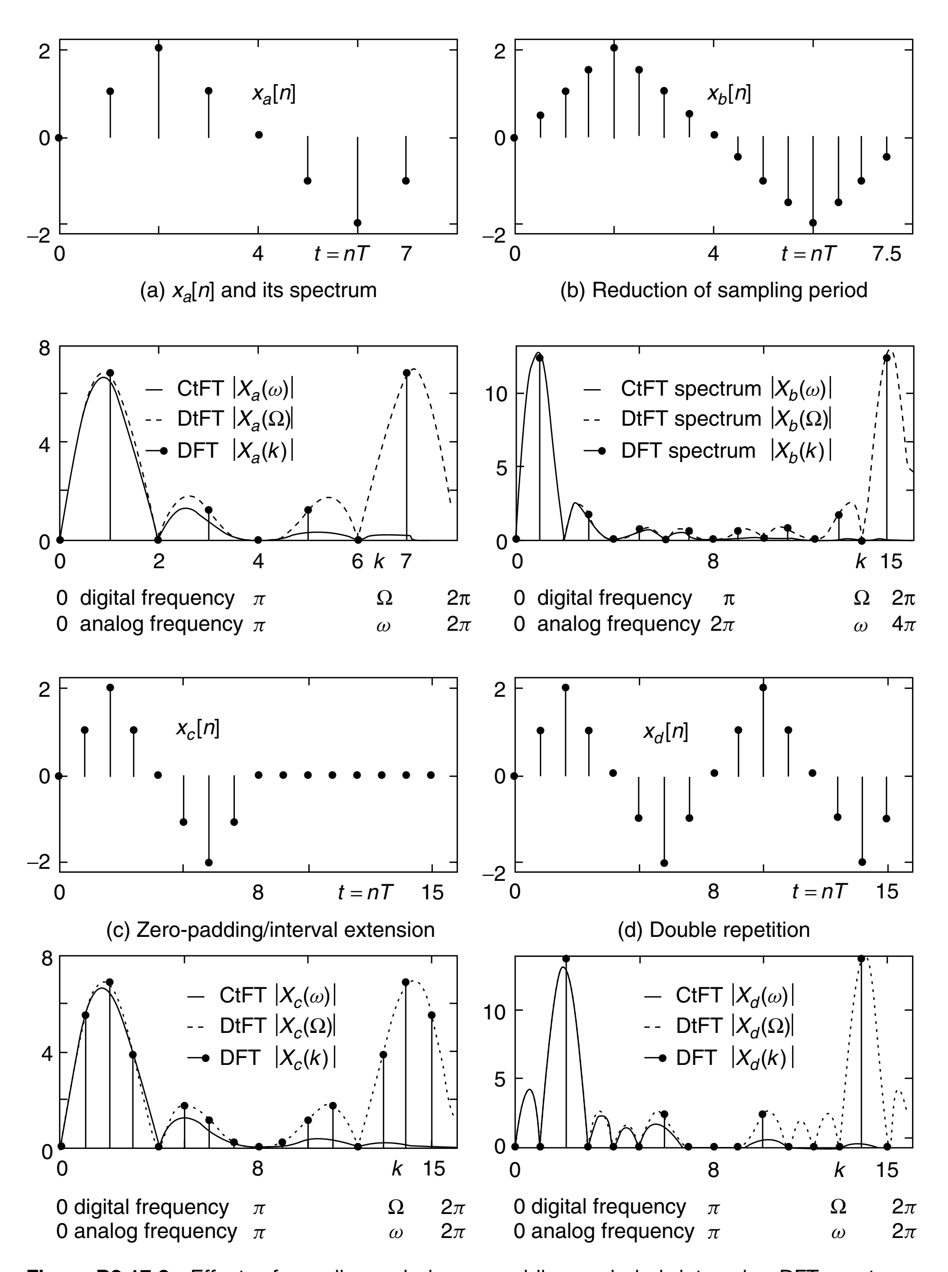


Figure P3.17.2 Effects of sampling period, zero-padding, and whole interval on DFT spectrum.

Successively, use the time shifting property (P3.17.3) to get the CtFT of

$$x(t) = \Lambda(t+2) - \Lambda(t-2) \tag{P3.17.5}$$

as

$$X(\omega) \overset{\text{(P3.17.3, 4)}}{=} T(\omega)e^{j2\omega} - T(\omega)e^{-j2\omega} = j8\sin(2\omega)\sin c^2\left(\frac{\omega}{\pi}\right) \tag{P3.17.6}$$

Get the CtFT $Y(\omega)$ of the triangular wave that is generated by repeating $x(t)$ two times and described as below.

$$y(t) = x(t+4) + x(t-4) \tag{P3.17.7}$$

Plot the spectrum $X(\omega)$ for $0 \le \omega \le 2\pi$ and check if the result is the same as depicted in a solid line in Fig. P3.17.2a or P3.17.2c. You can also plot the spectrum $X(\omega)$ for $0 \le \omega \le 4\pi$ and check if the result is the same as the solid line in Fig. P3.17.2b. Additionally, plot the spectrum $Y(\omega)$ for $0 \le \omega \le 2\pi$ and check if the result is the same as the solid line in Fig. P3.17.2d.

(b) The definition of DtFT, which is used for getting the spectrum of a discrete-time signal $x[n]$, is

$$X(\Omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\Omega n} \tag{P3.17.8}$$

Use this formula to compute the DtFTs of the discrete-time signals $x_a[n], x_b[n], x_c[n], x_d[n]$ and plot them to see if the results are the same as the dotted lines in Fig. P3.17.2a–d. What is the valid analog frequency range over which each DtFT spectrum is similar to the corresponding CtFT spectrum, respectively? Note that the valid analog frequency range is $[-\pi/T, +\pi/T]$ for the sampling period T.

(c) Use the definition (3.9.1a) of DFT to get the spectra of the discrete-time signals $x_a[n], x_b[n], x_c[n]$, and $x_d[n]$ and plot them to see if the results are the same as the dots in Fig. P3.17.2a–d. Do they match the samples of the corresponding DtFTs at $\Omega_k = 2k\pi/N$? Among the DFT spectra (a), (b), (c), and (d), which one describes the corresponding CtFT or DtFT spectra for the widest range of analog frequency?

3.18 Windowing Techniques Against the Leakage of DFT Spectrum

There are several window functions ready to be used for alleviating the spectrum leakage problem or for other purposes. We have made a MATLAB routine “`windowing()`” for easy application of the various windows.

Applying the Hamming window function to the discrete-time signal $x_d[n]$ in Fig. 3.13d, get the new DFT spectrum, plot its magnitude together with the windowed signal, check if they are the same as depicted in Fig. P3.18b, and compare it with the old DFT spectrum in Fig. 3.13d or Fig. P3.18a. You can start with the incomplete MATLAB program "nm3p18.m" below. What is the effect of windowing on the spectrum?

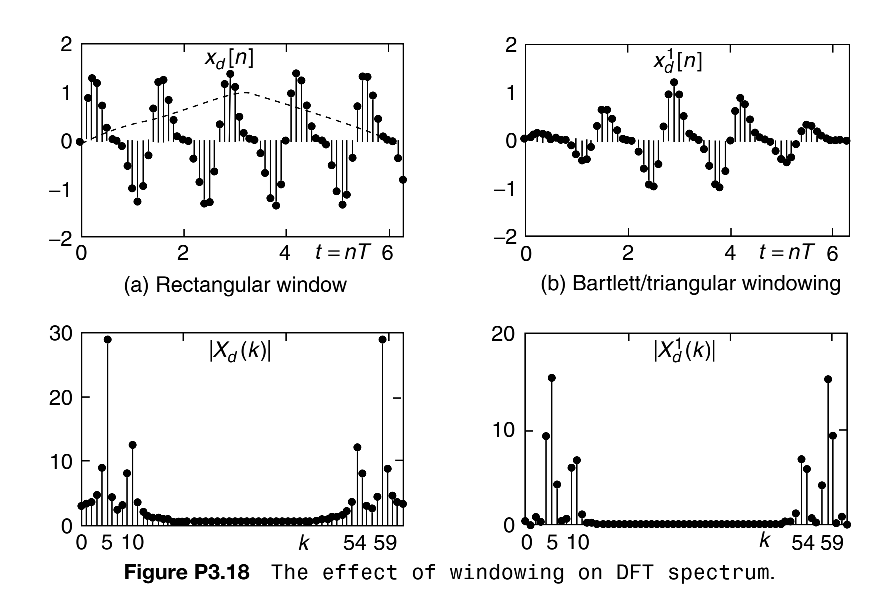


Figure P3.18 The effect of windowing on DFT spectrum.

```
function xw = windowing(x,w)
N = length(x);
if nargin < 2 | w == 'rt' | isempty(w), xw = x;
 elseif w == 'bt', xw = x.*bartlett(N)';
 elseif w == 'bk', xw = x.*blackman(N)';
 elseif w == 'hm', xw = x.*hamming(N)';
 end
```

```
%nm3p18: windowing effect on DFT spectrum
w1 = 1.5*pi; w2 = 3*pi; %two tones
N = 64; n = 1:N; T = 0.1; t = (n - 1)*T;
k = 1:N; w0 = 2*pi/T; w = (k - 1)*w0;
xbn = sin(w1*t) + 0.5*sin(w2*t);
xbwn = windowing(xbn,'bt');
Xb = fft(xbn); Xbw = fft(xbwn);
subplot(421), stem(t,xbn,'.')
subplot(423), stem(k,abs(Xb),'.')
..............
```

3.19 Interpolation by Using DFS: Zero-Padding on the Frequency Domain
The fitting curve in Fig. 3.14d has been obtained by zeroing out all the digital frequency components higher than $\pi/2$ [rad]($N/4 < k < 3N/4$) of the sequence $x[n]$ in Fig. 3.14a. Plot another fitting curve obtained by removing all the frequency components higher than $\pi/4$ [rad]($N/8 < k < 7N/8$) and compare it with Fig. 3.14d.

3.20 On-Line Recursive Computation of DFT

For the case where you need to compute the DFT of a block of data every time a new sampled data replaces the oldest one in the block, we derive the following recursive algorithm for DFT computation.

Defining the first data block and the mth data block as

$$\{x_0[0], x_0[1], \ldots, x_0[N-1]\} = \{0, 0, \ldots, 0\} \tag{P3.20.1}$$

$$\{x_m[0], x_m[1], \ldots, x_m[N-1]\} = \{x[m], x[m+1], \ldots, x[m+N-1]\} \tag{P3.20.2}$$

the DFT for the $(m+1)$th data block

$$\{x_{m+1}[0], x_{m+1}[1], \ldots, x_{m+1}[N-1]\} = \{x[m+1], x[m+2], \ldots, x[m+N]\} \tag{P3.20.3}$$

can be expressed in terms of the DFT for the mth data block

$$X_m(k) = \sum_{n=0}^{N-1} x_m[n]e^{-j2\pi nk/N}, \qquad k = 0 : N-1 \tag{P3.20.4}$$

as follows:

$$\begin{aligned}
X_{m+1}(k) &= \sum\nolimits_{n=0}^{N-1} x_{m+1}[n]e^{-j2\pi nk/N} = \sum\nolimits_{n=0}^{N-1} x_m[n+1]e^{-j2\pi nk/N} \\
&= \sum\nolimits_{n=0}^{N-1} x_m[n+1]e^{-j2\pi(n+1)k/N}e^{j2\pi k/N} \\
&= \sum\nolimits_{n=1}^{N} x_m[n]e^{-j2\pi nk/N}e^{j2\pi k/N} \\
&= \left\{\sum\nolimits_{n=0}^{N-1} x_m[n]e^{-j2\pi nk/N} + x[N] - x[0]\right\} e^{j2\pi k/N} \\
&= \{X_m(k) + x[N] - x[0]\}e^{j2\pi k/N}
\end{aligned} \tag{P3.20.5}$$

You can compute the 128-point DFT for a block composed of 128 random numbers by using this RDFT algorithm and compare it with that obtained

by using the MATLAB built-in routine "`fft()`". You can start with the incomplete MATLAB program "`do_RDFT.m`" below.

```
%do_RDFT
clear, clf
N = 128; k = [0:N - 1];
x = zeros(1,N); %initialize the data block
Xr = zeros(1,N); % and its DFT
for m = 0:N
   xN = rand; %new data
   Xr = (Xr + xN - x(1)).*???????????????? %RDFT formula (P3.20.5)
   x = [x(2:N) xN];
end
dif = norm(Xr-fft(x)) %difference between RDFT and FFT
```

4

NONLINEAR EQUATIONS

4.1 ITERATIVE METHOD TOWARD FIXED POINT

Let's see the following theorem.

Fixed-Point Theorem: ***Contraction Theorem***[K-2, Section 5.1]**.** Suppose a function $g(x)$ is defined and its first derivative $g'(x)$ exists continuously on some interval $I = [x^o - r, x^o + r]$ around the fixed point x^o of $g(x)$ such that

$$g(x^o) = x^o \tag{4.1.1}$$

Then, if the absolute value of $g'(x)$ is less than or equal to a positive number α that is strictly less than one, that is,

$$|g'(x)| \le \alpha < 1 \tag{4.1.2}$$

the iteration starting from any point $x_0 \in I$

$$x_{k+1} = g(x_k) \qquad \text{with } x_0 \in I \tag{4.1.3}$$

converges to the (unique) fixed point x^o of $g(x)$.

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

Proof. The Mean Value Theorem (MVT) (Appendix A) says that for any two points x_0 and x^o, there exists a point x between the two points such that

$$g(x_0) - g(x^o) = g'(x)(x_0 - x^o); \quad x_1 - x^o \overset{(4.1.3),(4.1.1)}{=} g'(x)(x_0 - x^o) \tag{1}$$

Taking the absolute value of both sides of (1) and using the precondition (4.1.2) yields

$$|x_1 - x^o| \le \alpha|x_0 - x^o| < |x_0 - x^o| \tag{2}$$

which implies that x_1 is closer to x^o than x_0 and thus still stays inside the interval I. Applying this successively, we can get

$$|x_k - x^o| \le \alpha|x_{k-1} - x^o| \le \alpha^2|x_{k-2} - x^o| \le \cdots \le \alpha^k|x_0 - x^o| \to 0 \text{ as } k \to \infty \tag{3}$$

which implies that the iterative sequence $\{x_k\}$ generated by (4.1.3) converges to x^o.

(Q) Is there any possibility that the fixed point is not unique—that is, more than one point satisfy Eq. (4.1.1) and so the iterative scheme may get confused among the several fixed points?

(A) It can never happen, because the points x^{o1} and x^{o2} satisfying Eq. (4.1.1) must be the same:

$$|x^{o1} - x^{o2}| = |g(x^{o1}) - g(x^{o2})| \le \alpha|x^{o1} - x^{o2}| \quad (\alpha < 1); \; |x^{o1} - x^{o2}| = 0; \; x^{o1} \equiv x^{o2}$$

In order to solve a nonlinear equation $f(x) = 0$ using the iterative method based on this fixed-point theorem, we must somehow arrange the equation into the form

$$x = g(x) \tag{4.1.4}$$

and start the iteration (4.1.3) with an initial value x_0, then continue until some stopping criterion is satisfied; for example, the difference $|x_{k+1} - x_k|$ between the successive iteration values becomes smaller than some predefined number (`TolX`) or the iteration number exceeds some predetermined number (`MaxIter`). This scheme is cast into the MATLAB routine "`fixpt()`". Note that the second output argument (`err`) is never the real error—that is, the distance to the true solution—but just the last value of $|x_{k+1} - x_k|$ as an error estimate. See the following remark and examples.

Remark 4.1. Fixed-Point Iteration. Noting that Eq. (4.1.4) is not unique for a given $f(x) = 0$, it would be good to have $g(x)$ such that $|g'(x)| < 1$ inside the interval I containing its fixed point x^o which is the solution we are looking for. It may not be so easy, however, to determine whether $|g'(x)| < 1$ is

satisfied around the solution point if we don't have any rough estimate of the solution.

```
function [x,err,xx] = fixpt(g,x0,TolX,MaxIter)
% solve x = g(x) starting from x0 by fixed-point iteration.
%input : g,x0 = the function and the initial guess
%        TolX = upperbound of incremental difference |x(n + 1) - x(n)|
%     MaxIter = maximum # of iterations
%output: x    = point which the algorithm has reached
%        err  = last value |x(k) - x(k - 1)| achieved
%        xx   = history of x
if nargin < 4, MaxIter = 100; end
if nargin < 3, TolX = 1e-6; end
xx(1) = x0;
for k = 2:MaxIter
   xx(k) = feval(g,xx(k - 1)); %Eq.(4.1.3)
   err = abs(xx(k) - xx(k - 1)); if err < TolX, break;  end
end
x = xx(k);
if k == MaxIter
  fprintf('Do not rely on me, though best in %d iterations\n',MaxIter)
end
```

Example 4.1. Fixed-Point Iteration. Consider the problem of solving the nonlinear equation

$$f_{41}(x) = x^2 - 2 = 0 \tag{E4.1.1}$$

In order to apply the fixed-point iteration for solving this equation, we need to convert it into a form like (4.1.4). Let's try with the following three forms and guess that the solution is in the interval $I = (1, 1.5)$.

(a) How about $x^2 - 2 = 0 \rightarrow x^2 = 2 \rightarrow x = 2/x = g_a(x)$? (E4.1.2)

Let's see if the absolute value of the first derivative of $g_a(x)$ is less than one for the solution interval, that is, $|g_a{}'(x)| = 2/x^2 < 1\,\forall\, x \in I$. This condition does not seem to be satisfied and so we must be pessimistic about the possibility of reaching the solution with (E4.1.2). We don't need many iterations to confirm this.

$$x_0 = 1;\; x_1 = \frac{2}{x_0} = 2;\; x_2 = \frac{2}{x_1} = 1;\; x_3 = \frac{2}{x_2} = 2;\; x_4 = \frac{2}{x_3} = 1; \cdots \tag{E4.1.3}$$

The iteration turned out to be swaying between 1 and 2, never approaching the solution.

(b) How about $x^2 - 2 = 0 \rightarrow (x-1)^2 + 2x - 3 = 0 \rightarrow x = -\frac{1}{2}\{(x-1)^2 - 3\} = g_b(x)$? (E4.1.4)

This form seems to satisfy the convergence condition

$$|g_b{}'(x)| = |x - 1| \le 0.5 < 1 \qquad \forall\, x \in I \tag{E4.1.5}$$

and so we may be optimistic about the possibility of reaching the solution with (E4.1.4). To confirm this, we need just a few iterations, which can be performed by using the routine "fixpt()".

```
>>gb=inline('-((x-1).^2-3)/2','x');
>>[x,err,xx]=fixpt(gb,1,1e-4,50);
>>xx
   1.0000   1.5000   1.3750   1.4297   1.4077   ...
```

The iteration is obviously converging to the true solution $\sqrt{2} = 1.414\ldots$, which we already know in this case. This process is depicted in Fig. 4.1a.

(c) How about $x^2 = 2 \rightarrow x = \frac{2}{x} \rightarrow x + x = \frac{2}{x} + x \rightarrow x = \frac{1}{2}\left(x + \frac{2}{x}\right) = g_c(x)$? (E4.1.6)

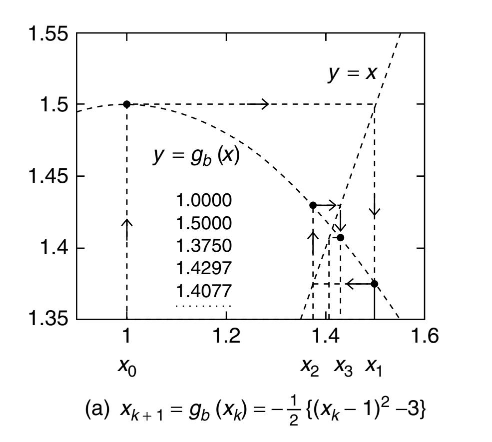
This form seems to satisfy the convergence condition

$$|g_c{}'(x)| = \frac{1}{2}\left|1 - \frac{2}{x^2}\right| \le 0.5 < 1 \qquad \forall x \in I \tag{E4.1.7}$$

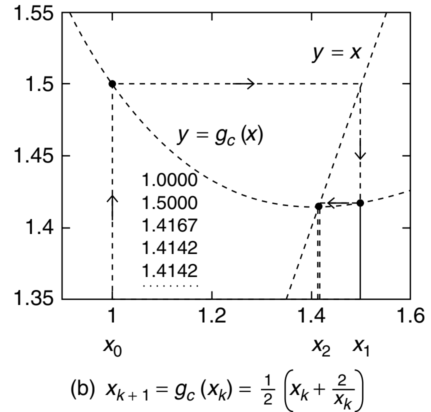
which guarantees that the iteration will reach the solution. Moreover, since this derivative becomes zero at the solution of $x^2 = 2$, we may expect fast convergence, which is confirmed by using the routine "fixpt()". The process is depicted in Fig. 4.1b.

```
>>gc = inline('(x+2./x)/2','x');
>>[x,err,xx] = fixpt(gc,1,1e-4,50);
>>xx
   1.0000   1.5000   1.4167   1.4142   1.4142   ...
```

(cf) In fact, if the nonlinear equation that we must solve is a polynomial equation, then it is convenient to use the MATLAB built-in command "roots()".



(a) $x_{k+1} = g_b(x_k) = -\frac{1}{2}\{(x_k - 1)^2 - 3\}$



(b) $x_{k+1} = g_c(x_k) = \frac{1}{2}\left(x_k + \frac{2}{x_k}\right)$

Figure 4.1 Iterative method to solve nonlinear equations based on the fixed-point theorem.

(Q) How do we make the iteration converge to another solution $x = -\sqrt{2}$ of $x^2 - 2 = 0$?

4.2 BISECTION METHOD

The bisection method can be applied for solving nonlinear equations like $f(x) = 0$, only in the case where we know some interval $[a, b]$ on which $f(x)$ is continuous and the solution uniquely exists and, most importantly, $f(a)$ and $f(b)$ have the opposite signs. The procedure toward the solution of $f(x) = 0$ is described as follows and is cast into the MATLAB routine "`bisct()`".

Step 0. Initialize the iteration number $k = 0$.
Step 1. Let $m = \frac{1}{2}(a + b)$. If $f(m) \approx 0$ or $\frac{1}{2}(b - a) \approx 0$, then stop the iteration.
Step 2. If $f(a)f(m) > 0$, then let $a \leftarrow m$; otherwise, let $b \leftarrow m$. Go back to step 1.

```
function [x,err,xx] = bisct(f,a,b,TolX,MaxIter)
%bisct.m to solve f(x) = 0 by using the bisection method.
%input : f   = ftn to be given as a string 'f' if defined in an M-file
%        a/b = initial left/right point of the solution interval
%       TolX = upperbound of error |x(k) - xo|
%    MaxIter = maximum # of iterations
%output: x   = point which the algorithm has reached
%        err = (b - a)/2(half the last interval width)
%        xx  = history of x
TolFun=eps; fa = feval(f,a); fb = feval(f,b);
if fa*fb > 0, error('We must have f(a)f(b)<0!'); end
for k = 1: MaxIter
  xx(k) = (a + b)/2;
  fx = feval(f,xx(k));  err = (b-a)/2;
   if abs(fx) < TolFun | abs(err)<TolX, break;
     elseif fx*fa > 0, a = xx(k); fa = fx;
     else  b = xx(k);
   end
end
x = xx(k);
if k == MaxIter, fprintf('The best in %d iterations\n',MaxIter), end
```

Remark 4.2. Bisection Method Versus Fixed-Point Iteration

1. Only if the solution exists on some interval $[a, b]$, the distance from the midpoint $(a + b)/2$ of the interval as an approximate solution to the true solution is at most one-half of the interval width—that is, $(b - a)/2$, which we take as a measure of error. Therefore, for every iteration of the bisection method, the upper bound of the error in the approximate solution decreases by half.

2. The bisection method and the false position method appearing in the next section will definitely give us the solution, only if the solution exists uniquely in some known interval. But the convergence of the fixed-point iteration depends on the derivative of $g(x)$ as well as the initial value x_0.
3. The MATLAB built-in routine `fzero(f,x)` finds a zero of the function given as the first input argument, based on the interpolation and the bisection method with the initial solution interval vector `x` $= [a\ b]$ given as the second input argument. The routine is supposed to work even with an initial guess `x` $= x_0$ of the (scalar) solution, but it sometimes gives us a wrong result as illustrated in the following example. Therefore, it is safe to use the routine `fzero()` with the initial solution interval vector $[a\ b]$ as the second input argument.

Example 4.2. Bisection Method. Consider the problem of solving the nonlinear equation

$$f_{42}(x) = \tan(\pi - x) - x = 0 \tag{E4.2.1}$$

Noting that $f_{42}(x)$ has the value of infinity at $x = \pi/2 = 1.57\ldots$, we set the initial solution interval to [1.6, 3] excluding the singular point and use the MATLAB routine "`bisct()`" as follows. The iteration seems to be converging to the solution as we expect (see Fig. 4.2b).

```
>>f42 = inline('tan(pi - x)-x','x');
>>[x,err,xx] = bisct(f42,1.6,3,1e-4,50);
>>xx
   2.3000   1.9500   2.1250   2.0375   1.9937   2.0156 ... 2.0287
```

But, if we start with the initial solution interval $[a, b]$ such that $f(a)$ and $f(b)$ have the same sign, we will face the error message.

```
>>[x,err,xx] = bisct(f42,1.5,3,1e-4,50);
   ??? Error using  ==> bisct
   We must have f(a)f(b)<0!
```

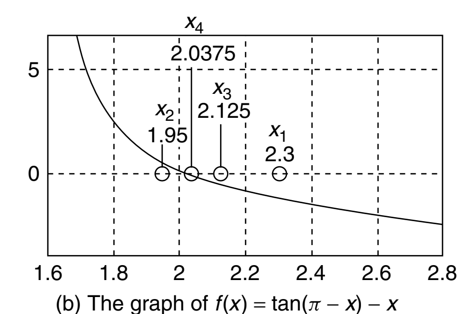
Now, let's see how the MATLAB built-in routine `fzero(f,x)` works.

```
>> fzero(f42,[1.6 3])
    ans = 2.0287 %good job!
>> fzero(f42,[1.5 3])
    ??? Error using  ==> fzero
    The function values at interval endpoints must differ in sign.
>> fzero(f42,1.8) %with an initial guess as 2nd input argument
    ans = 1.5708  %wrong result with no warning message
```

(cf) Not all the solutions given by computers are good, especially when we are careless.

k	a_k	x_k	b_k	$f(x_k)$
0	1.6		3.0	32.6, −2.86
1	1.6	2.3	3.0	−1.1808
2	1.6	1.95	2.3	0.5595
3	1.95	2.125	2.3	−0.5092
4	1.95	2.0375	2.125	−0.5027
⋮	⋮	⋮	⋮	⋮

(a) Process of the bisection method



(b) The graph of $f(x) = \tan(\pi - x) - x$

Figure 4.2 Bisection method for Example 4.2.

4.3 FALSE POSITION OR REGULA FALSI METHOD

Similarly to the bisection method, the false position or regula falsi method starts with the initial solution interval $[a, b]$ that is believed to contain the solution of $f(x) = 0$. Approximating the curve of $f(x)$ on $[a, b]$ by a straight line connecting the two points $(a, f(a))$ and $(b, f(b))$, it guesses that the solution may be the point at which the straight line crosses the x axis:

$$x = a - \frac{f(a)}{f(a) - f(b)}(b - a) = b - \frac{f(b)}{f(b) - f(a)}(b - a) = \frac{af(b) - bf(a)}{f(a) - f(b)} \tag{4.3.1}$$

```
function [x,err,xx] = falsp(f,a,b,TolX,MaxIter)
%bisct.m to solve f(x)=0 by using the false position method.
%input : f    = ftn to be given as a string 'f' if defined in an M-file
%         a/b = initial left/right point of the solution interval
%        TolX = upperbound of error(max(|x(k)-a|,|b-x(k)|))
%     MaxIter = maximum # of iterations
%output: x    = point which the algorithm has reached
%         err = max(x(last)-a|,|b-x(last)|)
%         xx  = history of x
TolFun = eps; fa = feval(f,a); fb=feval(f,b);
if fa*fb > 0, error('We must have f(a)f(b)<0!'); end
for k = 1: MaxIter
   xx(k) = (a*fb-b*fa)/(fb-fa); %Eq.(4.3.1)
   fx = feval(f,xx(k));
   err = max(abs(xx(k) - a),abs(b - xx(k)));
   if abs(fx) < TolFun | err<TolX, break;
    elseif fx*fa > 0, a = xx(k); fa = fx;
    else  b = xx(k); fb = fx;
   end
end
x = xx(k);
if k == MaxIter, fprintf('The best in %d iterations\n',MaxIter), end
```

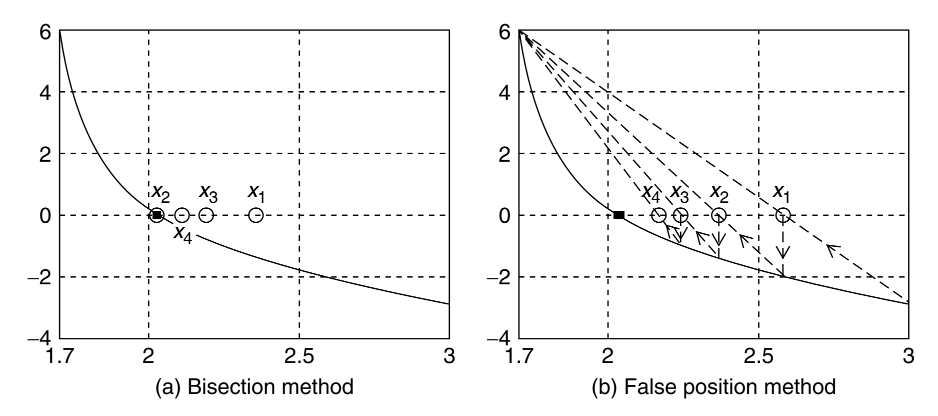


Figure 4.3 Solving the nonlinear equation $f(x) = \tan(\pi - x) - x = 0$.

For this method, we take the larger of $|x - a|$ and $|b - x|$ as the measure of error. This procedure to search for the solution of $f(x) = 0$ is cast into the MATLAB routine "`falsp()`".

Note that although the false position method aims to improve the convergence speed over the bisection method, it cannot always achieve the goal, especially when the curve of $f(x)$ on $[a, b]$ is not well approximated by a straight line as depicted in Fig. 4.3. Figure 4.3b shows how the false position method approaches the solution, started by typing the following MATLAB statements, while Fig. 4.3a shows the footprints of the bisection method.

```
>>[x,err,xx] = falsp(f42,1.7,3,1e-4,50) %with initial interval [1.7,3]
```

4.4 NEWTON(–RAPHSON) METHOD

Consider the problem of finding numerically one of the solutions, x^o, for a nonlinear equation

$$f(x) = (x - x^o)^m g(x) = 0$$

where $f(x)$ has $(x - x^o)^m (m$ is an even number) as a factor and so its curve is tangential to the x-axis without crossing it at $x = x^o$. In this case, the signs of $f(x^o - \varepsilon)$ and $f(x^o + \varepsilon)$ are the same and we cannot find any interval $[a, b]$ containing only x^o as a solution such that $f(a)f(b) < 0$. Consequently, bracketing methods such as the bisection or false position ones are not applicable to this problem. Neither can the MATLAB built-in routine `fzero()` be applied to solve as simple an equation as $x^2 = 0$, which you would not believe until you try it for yourself. Then, how do we solve it? The Newton(–Raphson) method can

be used for this kind of problem as well as general nonlinear equation problems, only if the first derivative of $f(x)$ exists and is continuous around the solution.

The strategy behind the Newton(–Raphson) method is to approximate the curve of $f(x)$ by its tangential line at some estimate x_k

$$y - f(x_k) = f'(x_k)(x - x_k) \tag{4.4.1}$$

and set the zero (crossing the x-axis) of the tangent line to the next estimate x_{k+1}.

$$0 - f(x_k) = f'(x_k)(x_{k+1} - x_k)$$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{4.4.2}$$

This Newton iterative formula is cast into the MATLAB routine "`newton()`", which is designed to generate the numerical derivative (Chapter 5) in the case where the derivative function is not given as the second input argument.

Here, for the error analysis of the Newton method, we consider the second-degree Taylor polynomial (Appendix A) of $f(x)$ about $x = x_k$:

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{f''(x_k)}{2}(x - x_k)^2$$

```
function [x,fx,xx] = newton(f,df,x0,TolX,MaxIter)
%newton.m  to solve f(x) = 0 by using Newton method.
%input:  f = ftn to be given as a string 'f' if defined in an M-file
%       df = df(x)/dx (If not given, numerical derivative is used.)
%       x0 = the initial guess of the solution
%     TolX = the upper limit of |x(k) - x(k-1)|
%  MaxIter = the maximum # of iteration
%output: x = the point which the algorithm has reached
%        fx = f(x(last)), xx = the history of x
h = 1e-4; h2 = 2*h; TolFun=eps;
if nargin == 4 & isnumeric(df), MaxIter = TolX; TolX = x0; x0 = df;  end
xx(1) = x0; fx = feval(f,x0);
for k = 1: MaxIter
  if ~isnumeric(df), dfdx = feval(df,xx(k)); %derivative function
   else dfdx = (feval(f,xx(k) + h)-feval(f,xx(k) - h))/h2; %numerical drv
  end
  dx = -fx/dfdx;
  xx(k+1) = xx(k)+dx; %Eq.(4.4.2)
  fx = feval(f,xx(k + 1));
  if abs(fx)<TolFun | abs(dx) < TolX, break; end
end
x = xx(k + 1);
if k == MaxIter, fprintf('The best in %d iterations\n',MaxIter), end
```

We substitute $x = x^o$ (the solution) into this and use $f(x^o) = 0$ to write

$$0 = f(x^o) \approx f(x_k) + f'(x_k)(x^o - x_k) + \frac{f''(x_k)}{2}(x^o - x_k)^2$$

and

$$-f(x_k) \approx f'(x_k)(x^o - x_k) + \frac{f''(x_k)}{2}(x^o - x_k)^2$$

Substituting this into Eq. (4.4.2) and defining the error of the estimate x_k as $e_k = x_k - x^o$, we can get

$$x_{k+1} \approx x_k + (x^o - x_k) + \frac{f''(x_k)}{2f'(x_k)}(x^o - x_k)^2,$$

$$|e_{k+1}| \approx \left|\frac{f''(x_k)}{2f'(x_k)}\right| e_k^2 = A_k e_k^2 = |A_k e_k||e_k| \tag{4.4.3}$$

This implies that once the magnitude of initial estimation error $|e_0|$ is small enough to make $|Ae_0| < 1$, the magnitudes of successive estimation errors get smaller very quickly so long as A_k does not become large. The Newton method is said to be 'quadratically convergent' on account of the fact that the magnitude of the estimation error is proportional to the square of the previous estimation error.

Now, it is time to practice using the MATLAB routine "`newton()`" for solving a nonlinear equation like that dealt with in Example 4.2. We have to type the following statements into the MATLAB command window.

```
>>x0 = 1.8; TolX = 1e-5; MaxIter = 50; %with initial guess 1.8,...
>>[x,err,xx] = newton(f42,x0,1e-5,50)  %1st order derivative
>>df42 = inline('-(sec(pi-x)).^2-1','x'); %1st order derivative
>>[x,err,xx1] = newton(f42,df42,1.8,1e-5,50)
```

Remark 4.3. Newton(–Raphson) Method

1. While bracketing methods such as the bisection method and the false position method converge in all cases, the Newton method is guaranteed to converge only in case where the initial value x_0 is sufficiently close to the solution x^o and $A(x) = |f''(x)/2f'(x)|$ is sufficiently small for $x \approx x^o$. Apparently, it is good for fast convergence if we have small $A(x)$—that is, the relative magnitude of the second-order derivative $|f''(x)|$ over $|f'(x)|$ is small. In other words, the convergence of the Newton method is endangered if the slope of $f(x)$ is too flat or fluctuates too sharply.
2. Note two drawbacks of the Newton(–Raphson) method. One is the effort and time required to compute the derivative $f'(x_k)$ at each iteration; the

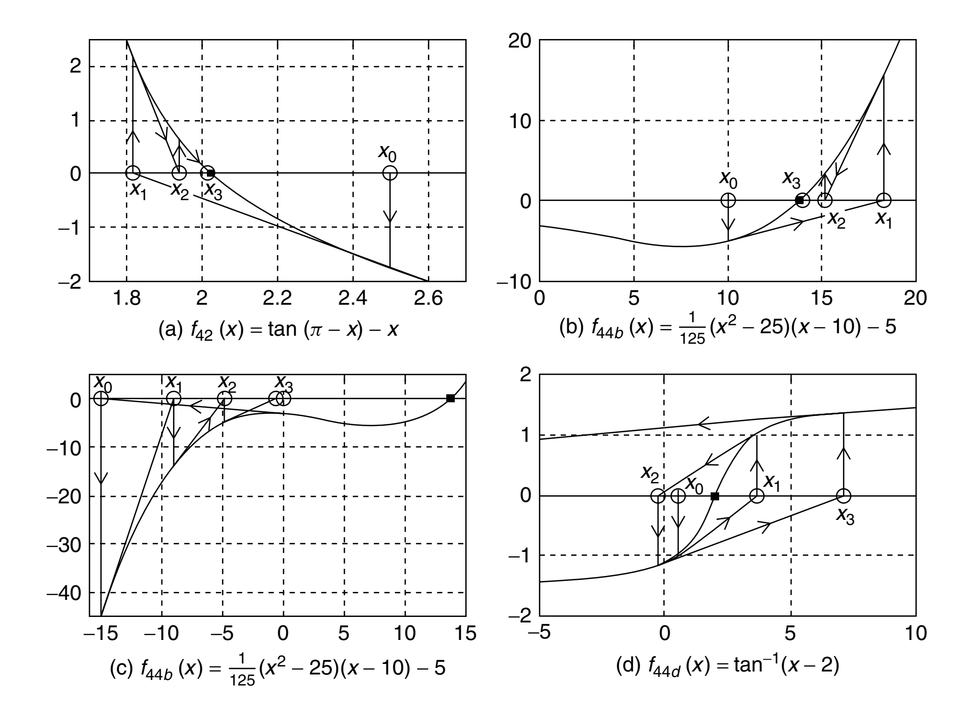


Figure 4.4 Solving nonlinear equations $f(x) = 0$ by using the Newton method.

other is the possibility of going astray, especially when $f(x)$ has an abruptly changing slope around the solution (e.g., Fig. 4.4c or 4.4d), whereas it converges to the solution quickly when $f(x)$ has a steady slope as illustrated in Figs. 4.4a and 4.4b.

4.5 SECANT METHOD

The secant method can be regarded as a modification of the Newton method in the sense that the derivative is replaced by a difference approximation based on the successive estimates

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \tag{4.5.1}$$

which is expected to take less time than computing the analytical or numerical derivative. By this approximation, the iterative formula (4.4.2) becomes

$$x_{k+1} = x_k - \frac{f(x_k)}{dfdx_k} \qquad \text{with } dfdx_k = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \tag{4.5.2}$$

```
function [x,fx,xx] = secant(f,x0,TolX,MaxIter,varargin)
% solve f(x) = 0 by using the secant method.
%input : f   = ftn to be given as a string 'f' if defined in an M-file
%        x0  = the initial guess of the solution
%        TolX  = the upper limit of |x(k) - x(k - 1)|
%        MaxIter = the maximum # of iteration
%output: x    = the point which the algorithm has reached
%        fx   = f(x(last)), xx = the history of x
h = 1e-4; h2 = 2*h; TolFun=eps;
xx(1) = x0;  fx = feval(f,x0,varargin{:});
for k = 1: MaxIter
   if k <= 1, dfdx = (feval(f,xx(k) + h,varargin{:})-...
                       feval(f,xx(k) - h,varargin{:}))/h2;
    else  dfdx = (fx - fx0)/dx;
   end
   dx = -fx/dfdx;
   xx(k + 1) = xx(k) + dx; %Eq.(4.5.2)
   fx0 = fx;
   fx = feval(f,xx(k+1));
   if abs(fx) < TolFun | abs(dx) < TolX,  break; end
end
x = xx(k + 1);
if k == MaxIter, fprintf('The best in %d iterations\n',MaxIter), end
```

This secant iterative formula is cast into the MATLAB routine “`secant()`”, which never needs anything like the derivative as an input argument. We can use this routine “`secant()`” to solve a nonlinear equation like that dealt with in Example 4.2, by typing the following statement into the MATLAB command window. The process is depicted in Fig. 4.5.

```
>>[x,err,xx] = secant(f42,2.5,1e-5,50) %with initial guess 1.8
```

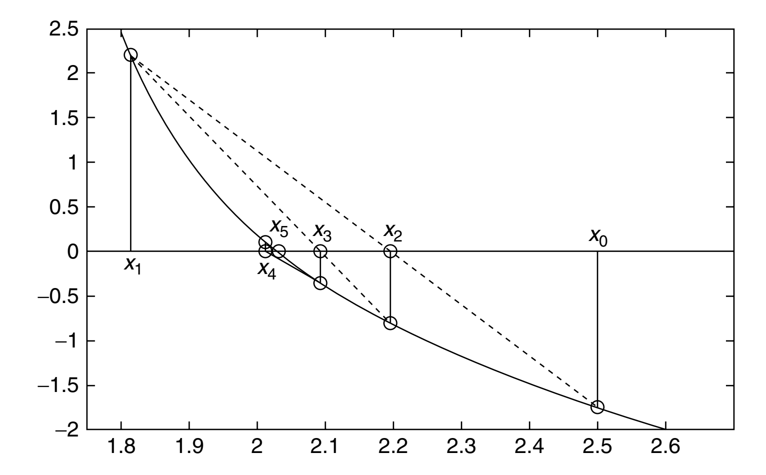


Figure 4.5 Solving a nonlinear equation by the secant method.

4.6 NEWTON METHOD FOR A SYSTEM OF NONLINEAR EQUATIONS

Note that the methods and the corresponding MATLAB routines mentioned so far can handle only one scalar equation with respect to one scalar variable. In order to see how a system of equations can be solved numerically, we rewrite the two equations

$$\begin{aligned} f_1(x_1, x_2) &= 0 \\ f_2(x_1, x_2) &= 0 \end{aligned} \tag{4.6.1}$$

by taking the Taylor series expansion up to first-order about some estimate point (x_{1k}, x_{2k}) as

$$\begin{aligned} f_1(x_1, x_2) &\cong f_1(x_{1k}, x_{2k}) + \left.\frac{\partial f_1}{\partial x_1}\right|_{(x_{1k}, x_{2k})} (x_1 - x_{1k}) + \left.\frac{\partial f_1}{\partial x_2}\right|_{(x_{1k}, x_{2k})} (x_2 - x_{2k}) = 0 \\ f_2(x_1, x_2) &\cong f_2(x_{1k}, x_{2k}) + \left.\frac{\partial f_2}{\partial x_1}\right|_{(x_{1k}, x_{2k})} (x_1 - x_{1k}) + \left.\frac{\partial f_2}{\partial x_2}\right|_{(x_{1k}, x_{2k})} (x_2 - x_{2k}) = 0 \end{aligned} \tag{4.6.2}$$

This can be arranged into a matrix–vector form as

$$\begin{aligned} \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} &\cong \begin{bmatrix} f_1(x_{1k}, x_{2k}) \\ f_2(x_{1k}, x_{2k}) \end{bmatrix} + \left.\begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{bmatrix}\right|_{(x_{1k}, x_{2k})} \begin{bmatrix} x_1 - x_{1k} \\ x_2 - x_{2k} \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned} \tag{4.6.3}$$

which we solve for (x_1, x_2) to get the updated vector estimate

$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \end{bmatrix} = \begin{bmatrix} x_{1k} \\ x_{2k} \end{bmatrix} - \left.\begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{bmatrix}\right|^{-1}_{(x_{1k}, x_{2k})} \begin{bmatrix} f_1(x_{1k}, x_{2k}) \\ f_2(x_{1k}, x_{2k}) \end{bmatrix} \tag{4.6.4}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - J_k^{-1}\mathbf{f}(\mathbf{x}_k) \quad \text{with the Jacobian } J_k(m, n) = [\partial f_m/\partial x_n]|_{\mathbf{x}_k}$$

This is not much different from the Newton iterative formula (4.4.2) and is cast into the MATLAB routine "`newtons()`". See Eq. (C.9) in Appendix C for the definition of the Jacobian.

Now, let's use this routine to solve the following system of nonlinear equations

$$\begin{aligned} x_1^2 + 4x_2^2 &= 5 \\ 2x_1^2 - 2x_1 - 3x_2 &= 2.5 \end{aligned} \tag{4.6.5}$$

In order to do so, we should first rewrite these equations into a form like Eq. (4.6.1) as

$$\begin{aligned} f_1(x_1, x_2) &= x_1^2 + 4x_2^2 - 5 = 0 \\ f_2(x_1, x_2) &= 2x_1^2 - 2x_1 - 3x_2 - 2.5 = 0 \end{aligned} \tag{4.6.6}$$

```
function [x,fx,xx] = newtons(f,x0,TolX,MaxIter,varargin)
%newtons.m  to solve a set of nonlinear eqs f1(x)=0, f2(x)=0,..
%input:  f    = 1^st-order vector ftn equivalent to a set of equations
%        x0   = the initial guess of the solution
%       TolX  = the upper limit of |x(k) - x(k - 1)|
%     MaxIter = the maximum # of iteration
%output: x    = the point which the algorithm has reached
%        fx   = f(x(last))
%        xx   = the history of x
h = 1e-4; TolFun = eps; EPS = 1e-6;
fx = feval(f,x0,varargin{:});
Nf = length(fx); Nx = length(x0);
if Nf ~= Nx, error('Incompatible dimensions of f and x0!'); end
if nargin < 4, MaxIter = 100; end
if nargin < 3, TolX = EPS; end
xx(1,:) = x0(:).'; %Initialize the solution as the initial row vector
%fx0 = norm(fx);                               %(1)
for k = 1: MaxIter
  dx = -jacob(f,xx(k,:),h,varargin{:})\fx(:);/;%-[dfdx]^-1*fx
  %for l = 1: 3 %damping to avoid divergence %(2)
  %dx = dx/2;                                %(3)
  xx(k + 1,:) = xx(k,:) + dx.';
  fx = feval(f,xx(k + 1,:),varargin{:}); fxn = norm(fx);
  % if fxn < fx0,  break;  end               %(4)
  %end                                       %(5)
  if fxn < TolFun | norm(dx) < TolX, break; end
  %fx0 = fxn;                                %(6)
end
x = xx(k + 1,:);
if k == MaxIter, fprintf('The best in %d iterations\n',MaxIter), end
```

```
function g = jacob(f,x,h,varargin) %Jacobian of f(x)
if nargin < 3,  h = 1e-4; end
h2 = 2*h;  N = length(x); x = x(:).'; I = eye(N);
for n = 1:N
  g(:,n) = (feval(f,x + I(n,:)*h,varargin{:}) ...
               -feval(f,x - I(n,:)*h,varargin{:}))'/h2;
end
```

and convert it into a MATLAB function defined in an M-file, say, "`f46.m`" as follows.

```
function y = f46(x)
y(1) = x(1)*x(1) + 4*x(2)*x(2) - 5;
y(2) = 2*x(1)*x(1)-2*x(1)-3*x(2) - 2.5;
```

Then, we type the following statements into the MATLAB command window:

```
>>x0 = [0.8 0.2]; x = newtons('f46',x0) %initial guess [.8 .2]
   x = 2.0000    0.5000
```

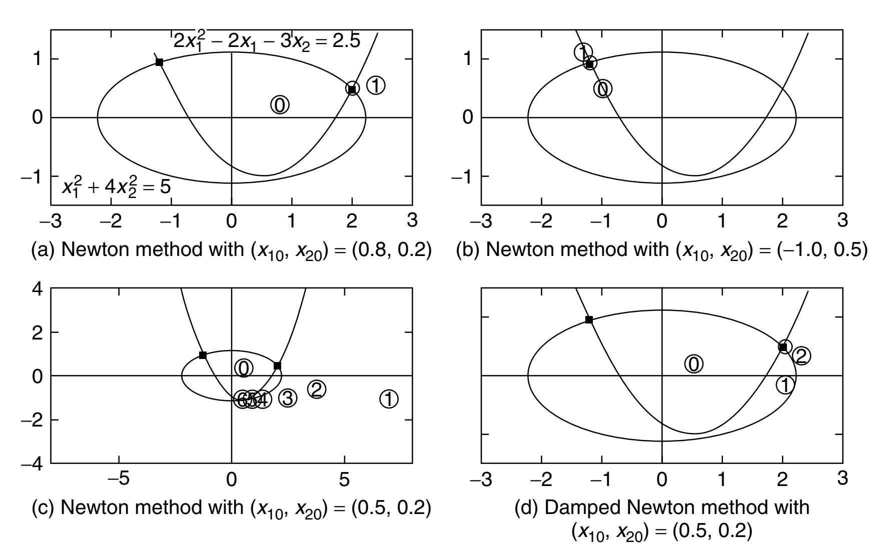


Figure 4.6 Solving the set (4.6.6) of nonlinear equations by vector Newton method.

Figure 4.6 shows how the vector Newton iteration may proceed depending on the initial guess (x_{10}, x_{20}). With $(x_{10}, x_{20}) = (0.8, 0.2)$, it converges to $(2, 0.5)$, which is one of the two roots (Fig. 4.6a) and with $(x_{10}, x_{20}) = (-1, 0.5)$, it converges to $(-1.2065, 0.9413)$, which is another root (Fig. 4.6b). However, with $(x_{10}, x_{20}) = (0.5, 0.2)$, it wanders around as depicted in Fig. 4.6c. From this figure, we can see that the iteration is jumping too far in the beginning and then going astray around the place where the curves of the two functions $f_1(x)$ and $f_2(x)$ are close, but not crossing. One idea for alleviating this problem is to modify the Newton algorithm in such a way that the step size can be adjusted (decreased) to keep the norm of $\mathbf{f}(\mathbf{x}_k)$ from increasing at each iteration. The so-called damped Newton method based on this idea will be implemented in the MATLAB routine "`newtons()`" if you activate the six statements numbered from 1 to 6 by deleting the comment mark(`%`) from the beginning of each line. With the same initial guess $(x_{10}, x_{20}) = (0.5, 0.2)$ as in Fig. 4.6c, the damped Newton method successfully leads to the point (2, 0.5), which is one of the two roots (Fig. 4.6d).

MATLAB has the built-in function "`fsolve(f,x0)`", which can give us a solution for a system of nonlinear equations. Let us try it for Eq. (4.6.5) or (4.6.6), which was already defined in the M-file named '`f46.m`'.

```
>>x = fsolve('f46',x0,optimset('fsolve')) %with default parameters
   x = 2.0000    0.5000
```

4.7 SYMBOLIC SOLUTION FOR EQUATIONS

MATLAB has many commands and functions that can be very helpful in dealing with complex analytic (symbolic) expressions and equations as well as in getting

numerical solutions. One of them is "`solve()`", which can be used for obtaining the symbolic or numeric roots of equations. According to what we could see by typing '`help solve`' into the MATLAB command window, its usages are as follows:

```
>>solve('p*sin(x) = r') %regarding x as an unknown variable and p as a parameter
    ans = asin(r/p)  %sin^-1(r/p)
>>[x1,x2] = solve('x1^2 + 4*x2^2 - 5 = 0','2* x 1^2 - 2*x1 - 3*x2-2.5 = 0')
    x1 = [                  2.]     x2 = [                0.500000]
         [           -1.206459]          [                0.941336]
         [0.603229 -0.392630*i]          [-1.095668 -0.540415e-1*i]
         [0.603229 +0.392630*i]          [-1.095668 +0.540415e-1*i]
>>S = solve('x^3 - y^3 = 2','x = y') %returns the solution in a structure.
  S = x: [3x1 sym]
      y: [3x1 sym]
>>S.x
  ans = [                  1]
        [ -1/2+1/2*i*3^(1/2)]
        [ -1/2-1/2*i*3^(1/2)]
>>S.y
  ans = [                 -1]
        [ 1/2 - 1/2*i*3^(1/2)]
        [ 1/2 + 1/2*i*3^(1/2)]
>>[u,v] = solve('a*u^2 + v^2 = 0','u - v = 1')%regarding u,v as unknowns and a as a parameter
 u = [1/2/(a + 1)*(-2*a + 2*(-a)^(1/2)) + 1] v = [1/2/(a + 1)*(-2*a + 2*(-a)^(1/2))]
     [1/2/(a + 1)*(-2*a - 2*(-a)^(1/2)) + 1]     [1/2/(a + 1)*(-2*a - 2*(-a)^(1/2))]
>>[a,u] = solve('a*u^2 + v^2','u-v = 1','a,u') %regards only v as a parameter
  a = -v^2/(v^2 + 2*v + 1)                u = v + 1
```

Note that in the case where the routine "`solve()`" finds the symbols more than the equations in its input arguments—say, M symbols and N equations with $M > N$—it regards the N symbols closest alphabetically to '`x`' as variables and the other $M - N$ symbols as constants, giving the priority of being a variable to the symbol after '`x`' than to one before '`x`' for two symbols that are at the same distance from '`x`'. Consequently, the priority order of being treated as a symbolic variable is as follows:

$$x > y > w > z > v > u > t > s > r > q > \cdots$$

Actually, we can use the MATLAB built-in function "`findsym()`" to see the priority order.

```
>>syms x y z q r s t u v w %declare 10 symbols to consider
>>findsym(x + y + z*q*r + s + t*u - v - w,10) %symbolic variables?
  ans = x,y,w,z,v,u,t,s,r,q
```

4.8 A REAL-WORLD PROBLEM

Let's see the following example.

Example 4.3. The Orbit of NASA's "Wind" Satellite. One of the previous NASA plans is to launch a satellite, called Wind, which is to stay at a fixed position along a line from the earth to the sun as depicted in Fig. 4.7 so that the solar wind passes around the satellite on its way to earth. In order to find the distance

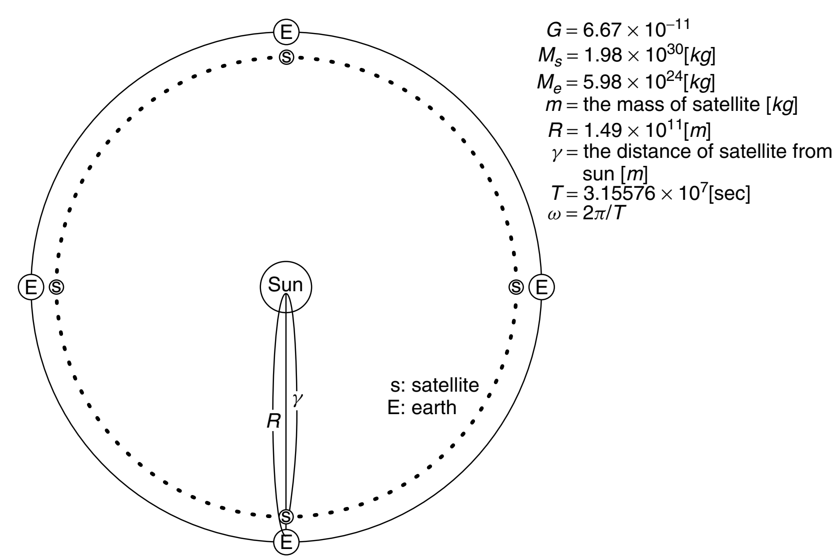


Figure 4.7 The orbit of a satellite.

of the satellite from earth, we set up the following equation based on the related physical laws as

$$G\frac{M_s m}{r^2} = G\frac{M_e m}{(R-r)^2} + mr\omega^2 \rightarrow G\left(\frac{M_S}{r^2} - \frac{M_e}{(R-r)^2}\right) - r\omega^2 = 0 \quad \text{(E4.3.1)}$$

(a) This might be solved for r by using the (nonlinear) equation solvers like the routine ‘`newtons()`’ (Section 4.6) or the MATLAB built-in routine ‘`fsolve()`’. We define this residual error function (whose zero is to be found) in the M-file named “`phys.m`” and run the statements in the following program “`nm4e03.m`” as

```
x0 = 1e6; %the initial (starting) guess
rn = newtons('phys',x0,1e-4,100) % newtons()
rfs = fsolve('phys',x0,optimset('fsolve')) % fsolve()
rfs1 = fsolve('phys',x0,optimset('MaxFunEvals',1000)) %more iterations
x01 = 1e10 %with another starting guess closer to the solution
rfs2 = fsolve('phys',x01,optimset('MaxFunEvals',1000))
residual_errrs = phys([rn rfs rfs1 rfs2])
```

which yields

```
rn   = 1.4762e+011 <with residual error of -1.8908e-016>
rfs  = 5.6811e+007 <with residual error of  4.0919e+004>
rfs1 = 2.1610e+009 <with residual error of  2.8280e+001>
rfs2 = 1.0000e+010 <with residual error of  1.3203e+000>
```

It seems that, even with the increased number of function evaluations and another initial guess as suggested in the warning message, ‘`fsolve()`’ is not so successful as ‘`newtons()`’ in this case.

(b) Noting that Eq. (E4.3.1) may cause 'division-by-zero', we multiply both sides of the equation by $r^2(R-r)^2$ to rewrite it as

$$r^3(R-r)^2\omega^2 - GM_S(R-r)^2 + GM_e r^2 = 0 \tag{E4.3.2}$$

We define this residual error function in the M-file named "physb.m" and run the following statements in the program "nm4e03.m":

```
rnb = newtons('physb',x0)
rfsb = fsolve('physb',x0,optimset('fsolve'))
residual_errs = phys([rnb rfsb])
```

which yields

```
rnb  = 1.4762e+011 <with residual error of 4.3368e-018>
rfsb = 1.4762e+011 <with residual error of 4.3368e-018>
```

Both of the two routines 'newtons()' and 'fsolve()' benefited from the function conversion and succeeded in finding the solution.

(c) The results obtained in (a) and (b) imply that the performance of the nonlinear equation solvers may depend on the shape of the (residual error) function whose zero they aim to find. Here, we try applying them with scaling. On the assumption that the solution is known to be on the order of 10^{11}, we divide the unknown variable r by 10^{11} to scale it down into the order of one. This can be done by substituting $r = r'/10^{11}$ into the equations and multiplying the resulting solution by 10^{11}. We can run the following statements in the program "nm4e03.m":

```
scale = 1e11;
rns = newtons('phys',x0/scale,1e-6,100,scale)*scale
rfss = fsolve('phys',x0/scale,optimset('fsolve'),scale)*scale
residual_errs = phys([rns rfss])
```

which yields

```
rns  = 1.4762e+011 <with residual error of -6.4185e-016>
rfss = 1.4763e+011 <with residual error of -3.3365e-006>
```

Compared with the results with no scaling obtained in (a), the routine 'fsolve()' benefited from scaling and succeeded in finding the solution.

(cf) This example implies the following tips for solving nonlinear equations.

- If you have some preliminary knowledge about the approximate value of the true solution, scale the unknown variable up/down to around one and then scale the resulting solution back down/up to get the solution to the original equation.
- It might be better for you to apply at least two methods to solve the equations as a cross-check. It is suggested to use 'newtons()' together with 'fsolve()' for confirming the solution of a system of nonlinear equations.

```
%nm4e03 – astrophysics
clear, clf
global G Ms Me R T
G = 6.67e11; Ms = 1.98e30; Me = 5.98e24;
R = 1.49e11; T = 3.15576e7;  w = 2*pi/T;
x0 = 1e6 %initial guess
format short e
disp('(a)')
rn = newtons('phys',x0)
rfs = fsolve('phys',x0 ,optimset('fsolve'))
%fsolve('phys',x0)/fsolve('phys',x0,foptions) in MATLAB 5.x version
rfs1=fsolve('phys',x0,optimset('MaxFunEvals',1000)) %more iterations
%options([2 3 14])=[1e-4 1e-4 1000];
%fsolve('phys',x0,options) in MATLAB 5.x
x01 = 1e10; %with another starting guess closer to the solution
rfs2 = fsolve('phys',x01,optimset('MaxFunEvals',1000))
residual_errs = phys([rn rfs rfs1 rfs2])
disp('(b)')
rnb = newtons('physb',x0)
rfsb = fsolve('physb',x0,optimset('fsolve'))
residual_errs = phys([rnb rfsb])
disp('(c)')
scale = 1e11;
rns = newtons('phys',x0/scale,1e-6,100,scale)*scale;
rfss = fsolve('phys',x0/scale,optimset('fsolve'),scale)*scale
residual_errs = phys([rns rfss])
```

```
function f = phys(x,scale);
if nargin < 2, scale = 1; end
global G Ms Me R T
w = 2*pi/T; x = x*scale; f = G*(Ms/(x.^2 + eps) - Me./((R - x).^2 + eps))-x*w^2;
```

```
function f = physb(x,scale);
if nargin < 2, scale = 1; end
global G Ms Me R T
w = 2*pi/T; x = x*scale; f = (R-x).^2.*(w^2*x.^3 - G*Ms) + G*Me*x.^2;
```

PROBLEMS

4.1 Fixed-Point Iterative Method

Consider the simple nonlinear equation

$$f(x) = x^2 - 3x + 1 = 0 \tag{P4.1.1}$$

Knowing that this equation has two roots

$$x^o = 1.5 \pm \sqrt{1.25} \approx 2.6180 \text{ or } 0.382; \quad x^{o1} \approx 0.382,\ x^{o2} \approx 2.6180 \tag{P4.1.2}$$

investigate the practicability of the fixed-point iteration.

(a) First consider the following iterative formula:

$$x_{k+1} = g_a(x_k) = \frac{1}{3}(x_k^2 + 1) \tag{P4.1.3}$$

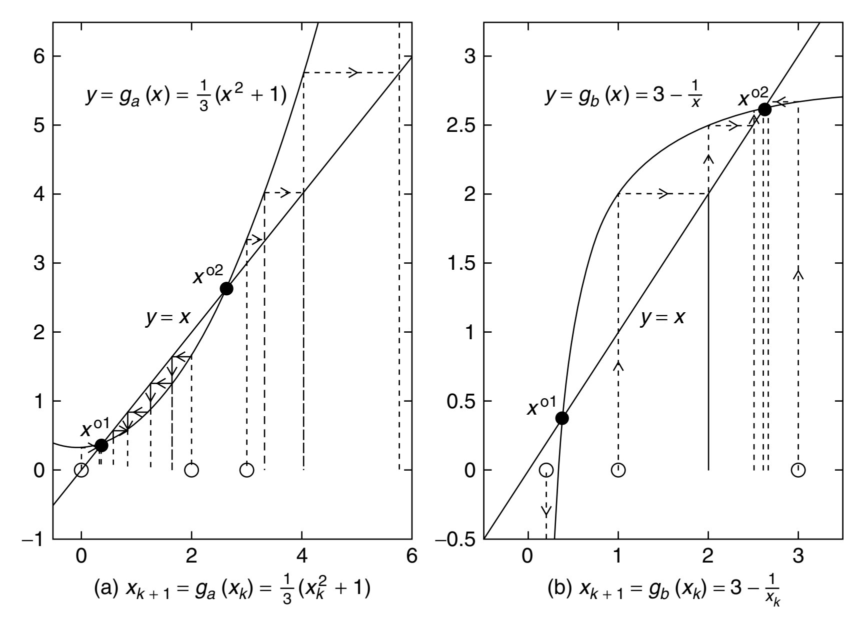


Figure P4.1 Iterative method based on the fixed-point theorem.

Noting that the first derivative of this iterative function $g_a(x)$ is

$$g_a'(x) = \frac{2}{3}x \tag{P4.1.4}$$

determine which solution attracts this iteration and certify it in Fig. P4.1a. In addition, run the MATLAB routine "`fixpt()`" to perform the iteration (P4.1.3) with the initial points $x_0 = 0$, $x_0 = 2$, and $x_0 = 3$. What does the routine yield for each initial point?

(b) Now consider the following iterative formula:

$$x_{k+1} = g_b(x_k) = 3 - \frac{1}{x_k} \tag{P4.1.5}$$

Noting that the first derivative of this iterative function $g_b(x)$ is

$$g_b'(x) = -\frac{1}{x^2} \tag{P4.1.6}$$

determine which solution attracts this iteration and certify it in Fig. P4.1b. In addition, run the MATLAB routine "`fixpt()`" to carry out the iteration (P4.1.5) with the initial points $x_0 = 0.2$, $x_0 = 1$, and $x_0 = 3$. What does the routine yield for each initial point?

(cf) This illustrates that the outcome of an algorithm may depend on the starting point.

4.2 Bisection Method and Fixed-Point Iteration

Consider the nonlinear equation treated in Example 4.2.

$$f(x) = \tan(\pi - x) - x = 0 \tag{P4.2.1}$$

Two graphical solutions of this equation are depicted in Fig. P4.2, which can be obtained by typing the following statements into the MATLAB command window:

```
>>ezplot('tan(pi-x)',-pi/2,3*pi/2)
>>hold on, ezplot('x+0',-pi/2,3*pi/2)
```

(a) In order to use the bisection method for finding the solution between 1.5 and 3, Charley typed the statements shown below. Could he get the right solution? If not, explain him why he failed and suggest him how to make it.

```
>>fp42 = inline('tan(pi-x)-x','x');
>>TolX = 1e-4; MaxIter = 50;
>>x = bisct(fp42,1.5,3,TolX,MaxIter)
```

(b) In order to find some interval to which the bisection method is applicable, Jessica used the MATLAB command "`find()`" as shown below.

```
>>x = [0: 0.5: pi]; y = tan(pi-x) - x;
>>k = find(y(1:end-1).*y(2:end) < 0);
>>[x(k) x(k + 1); y(k) y(k + 1)]
  ans =    1.5000    2.0000    2.0000    2.5000
         -15.6014    0.1850    0.1850   -1.7530
```

This shows that the sign of $f(x)$ changes between $x = 1.5$ and 2.0 and also between $x = 2.0$ and 2.5. Noting this, Jessica thought that she might use the bisection method to find a solution between 1.5 and 2.0 by typing the following command.

```
>>x=bisct(fp42,1.5,2,TolX,MaxIter)
```

Check the validity of the solution—that is, check if $f(x) = 0$ or not—by typing

```
>>fp42(x)
```

If her solution is not good, explain the reason. If you are not sure about it, you can try plotting the graph in Fig. P4.2 by typing the following statements into the MATLAB command window.

```
>>x = [-pi/2+0.05:0.05:3*pi/2 - 0.05];
>>plot(x,tan(pi - x),x,x)
```

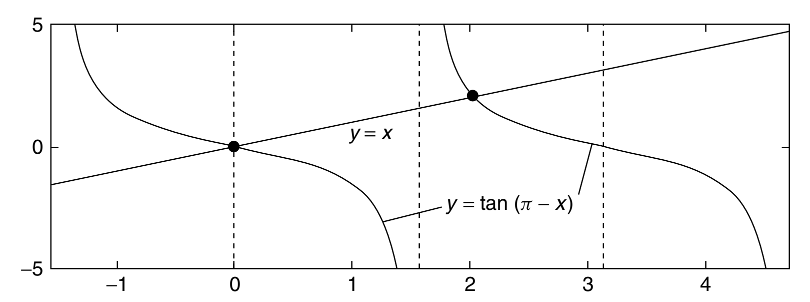


Figure P4.2 The graphical solutions of $\tan(\pi - x) - x = 0$ or $\tan(\pi - x) = x$.

(cf) This helps us understand why `fzero(fp42,1.8)` leads to the wrong solution even without any warning message as mentioned in Example 4.2.

(c) In order to find the solution around $x = 2.0$ by using the fixed-point iteration with the initial point $x_0 = 2.0$, Vania defined the iterative function as

```
>>gp421 = inline('tan(pi - x)','x'); % x = g1(x) = tan (π − x)
```

and typed the following statement into the MATLAB command window.

```
>>x = fixpt(gp421,2,TolX,MaxIter)
```

Could she reach the solution near 2? Will it be better if you start the routine with any different initial point? What is wrong?

(d) Itha, seeing what Vania did, decided to try with another iterative formula

$$\tan^{-1} x = \pi, \qquad x = g_2(x) = \pi - \tan^{-1} x \tag{P4.2.2}$$

So she defined the iterative function as

```
>>gp422 = inline('pi-atan(x)', 'x'); % x = g(x) = π − tan^-1(x)
```

and typed the following statement into the MATLAB command window:

```
>>x = fixpt(gp422,2,TolX,MaxIter)
```

What could she get? Is it the right solution? Does this command work with different initial value, like 0 or 6, which are far from the solution we want to find? Describe the difference between Vania's approach and Itha's.

4.3 Recursive (Self-Calling) Routine for Bisection Method

As stated in Section 1.3, MATLAB allows us to make nested (recursive) routines which call itself. Modify the MATLAB routine "`bisct()`" (in Section 4.2) into a nested routine "`bisct_r()`" and run it to solve Eq. (P4.2.1).

4.4 Newton Method and Secant Method

As can be seen in Fig. 4.5, the secant method introduced in Section 4.5 was devised to remove the necessity of the derivative/gradient and improve the convergence. But, it sometimes turns out to be worse than the Newton method. Apply the routines "`newton()`" and "`secant()`" to solve

$$f_{p44}(x) = x^3 - x^2 - x + 1 = 0 \tag{P4.4}$$

starting with the initial point $x_0 = -0.2$ one time and $x_0 = -0.3$ for another shot.

4.5 Acceleration of Aitken–Steffensen Method

A sequence converging to a limit x^o can be described as

$$x^o - x_{k+1} = e_{k+1} \approx Ae_k = A(x^o - x_k)$$

$$\text{with } \lim_{k\to\infty} \frac{x^o - x_{k+1}}{x^o - x_k} = A(|A| < 1) \tag{P4.5.1}$$

In order to think about how to improve the convergence speed of this sequence, we define a new sequence p_k as

$$\frac{x^o - x_{k+1}}{x^o - x_k} \approx A \approx \frac{x^o - x_k}{x^o - x_{k-1}};\ (x^o - x_{k+1})(x^o - x_{k-1}) \approx (x^o - x_k)^2$$

$$(x^o)^2 - x_{k+1}x^o - x_{k-1}x^o + x_{k+1}x_{k-1} \approx (x^o)^2 - 2x^o x_k + x_k^2$$

$$x^o \approx \frac{x_{k+1}x_{k-1} - x_k^2}{x_{k+1} - 2x_k + x_{k-1}} = p_k \tag{P4.5.2}$$

(a) Check that the error of this sequence p_k is as follows.

$$\begin{aligned}
x^o - p_k &= x^o - \frac{x_{k+1}x_{k-1} - x_k^2}{x_{k+1} - 2x_k + x_{k-1}} \\
&= x^o - \frac{x_{k-1}(x_{k+1} - 2x_k + x_{k-1}) - x_{k-1}^2 + 2x_{k-1}x_k - x_k^2}{x_{k+1} - 2x_k + x_{k-1}} \\
&= x^o - x_{k-1} + \frac{(x_k - x_{k-1})^2}{x_{k+1} - 2x_k + x_{k-1}} \\
&= x^o - x_{k-1} + \frac{(-(x^o - x_k) + (x^o - x_{k-1}))^2}{-(x^o - x_{k+1}) + 2(x^o - x_k) - (x^o - x_{k-1})} \\
&= x^o - x_{k-1} + \frac{(-A + 1)^2(x^o - x_{k-1})^2}{(-A^2 + 2A - 1)(x^o - x_{k-1})} = 0
\end{aligned} \tag{P4.5.3}$$

Table P4.5 Comparison of Various Methods Applied for Solving Nonlinear Equations

		Newton	Secant	Steffensen	Schroder	`fzero()`	`fsolve()`
$x_0 = 1.6$	x	2.0288					
f_{42}	$f(x)$		1.19e-8				1.72e-9
	Flops	158	112	273	167	986	1454
$x_0 = 0$	x			1.0000			
f_{p44}	$f(x)$						
	Flops	53	30	63	31	391	364
$x_0 = 0$	x			5.0000		NaN	
f_{p45}	$f(x)$					NaN	
	Flops	536	434	42	19	3683	1978

(cf) Since the `flops()` command is no longer available in MATLAB 6.x version, the numbers of floating-point operations are obtained from MATLAB 5.x version so that the readers can compare the various algorithms in terms of their computational loads.

(b) Modify the routine "`newton()`" into a routine "`stfns()`" that generates the sequence (P4.5.2) and run it to solve

$$f_{42}(x) = \tan(\pi - x) - x = 0 \quad (\text{with } x_0 = 1.6) \tag{P4.5.4}$$

$$f_{p44}(x) = x^3 - x^2 - x + 1 = 0 \quad (\text{with } x_0 = 0) \tag{P4.5.5}$$

$$f_{p45}(x) = (x - 5)^4 = 0 \quad (\text{with } x_0 = 0) \tag{P4.5.6}$$

Fill in Table P4.5 with the results and those obtained by using the routines "`newton()`", "`secant()`" (with the error tolerance `TolX` $= 10^{-5}$), "`fzero()`", and "`fsolve()`".

4.6 Acceleration of Newton Method for Multiple Roots: Schroder Method

In order to improve the convergence speed, Schroder modifies the Newton iterative algorithm (4.4.2) as

$$x_{k+1} = x_k - M\frac{f(x_k)}{f'(x_k)} \tag{P4.6.1}$$

with M : the order of multiplicity of the root we want to find

Based on this idea, modify the routine "`newton()`" into a routine "`schroder()`" and run it to solve Eqs. (P4.5.4.6). Fill in the corresponding blanks of Table P4.5 with the results.

4.7 Newton Method for Systems of Nonlinear Equations

Apply the routine "`newtons()`" (Section 4.6) and the MATLAB built-in routine "`fsolve()`" (with [x0 y0] = [1 0.5]) to solve the following systems of equations. Fill in Table P4.7 with the results.

(a) $$x^2 + y^2 = 1 \\ x^2 - y = 0 \quad \text{(P4.7.1)}$$

(b) $$5\cos\theta_1 + 6\cos(\theta_1 + \theta_2) = 10 \\ 5\sin\theta_1 + 6\sin(\theta_1 + \theta_2) = 4 \quad \text{(P4.7.2)}$$

(c) $$3x^2 + 4y^2 = 3 \\ x^2 + y^2 = \sqrt{3}/2 \quad \text{(P4.7.3)}$$

(d) $$x_1^3 + 10x_1 - x_2 = 5 \\ x_1 + x_2^3 - 10x_2 = -1 \quad \text{(P4.7.4)}$$

(e) $$x^2 - \sqrt{3}xy + 2y^2 = 10 \\ 4x^2 + 3\sqrt{3}xy + y = 22 \quad \text{(P4.7.5)}$$

(f) $$x^3y - y - 2x^3 = -16 \\ x - y^2 = -1 \quad \text{(P4.7.6)}$$

(g) $$x^2 + 4y^2 = 16 \\ xy^2 = 4 \quad \text{(P4.7.7)}$$

(h) $$xe^y - x^5 + y = 3 \\ x + y + \tan x - \sin y = 0 \quad \text{(P4.7.8)}$$

(i) $$2\log y - x = 0 \\ xy - y = 1 \quad \text{(P4.7.9)}$$

(j) $$12xy - 6x = -1 \\ 60x^2 - 180x^2y - 30xy = 1 \quad \text{(P4.7.10)}$$

4.8 Newton Method for Systems of Nonlinear Equations

Apply the routine "`newtons()`" (Section 4.6) and the MATLAB built-in routine "`fsolve()`" (with [x0 y0 z0] = [1 1 1]) to solve the following systems of equations. Fill in Table P4.8 with the results.

(a) $$xyz = -1 \\ x^2 + 2y^2 + 4z^2 = 7 \\ 2x^2 + y^3 + 6z = 7 \quad \text{(P4.8.1)}$$

(b) $$xyz = 1 \\ x^2 + 2y^3 + z^2 = 4 \\ x + 2y^2 - z^3 = 2 \quad \text{(P4.8.2)}$$

(c) $$x^2 + 4y^2 + 9z^2 = 34 \\ x^2 + 9y^2 - 5z = 40 \\ x^2z - y = 7 \quad \text{(P4.8.3)}$$

(d) $$x^2 + 2\sin(y\pi/2) + z^2 = 0 \\ -2xy + z = 3 \\ e^{x+y} - z^2 = 0 \quad \text{(P4.8.4)}$$

Table P4.7 Applying `newtons()`/`fsolve()` for Systems of Nonlinear Equations

		`newtons()`	`fsolve()`
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.1)	$\mathbf{x}$		
	$\|\|f(\mathbf{x})\|\|$		
	Flops	1043	1393
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.2)	$\mathbf{x}$	[0.1560 0.4111]	
	$\|\|f(\mathbf{x})\|\|$	3.97e-15 (3.66e-15)	
	Flops	2489	3028
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.3)	$\mathbf{x}$		
	$\|\|f(\mathbf{x})\|\|$		
	Flops	1476	3821
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.4)	$\mathbf{x}$		[0.5024 0.1506]
	$\|\|f(\mathbf{x})\|\|$		8.88e-16 (1.18e-6)
	Flops	1127	1932
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.5)	$\mathbf{x}$		
	$\|\|f(\mathbf{x})\|\|$		
	Flops	2884	3153
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.6)	$\mathbf{x}$	[1.6922 -1.6408]	
	$\|\|f(\mathbf{x})\|\|$	1.83e-15	
	Flops	9234	12896
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.7)	$\mathbf{x}$		
	$\|\|f(\mathbf{x})\|\|$		
	Flops	2125	2378
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.8)	$\mathbf{x}$		[0.2321 1.5067]
	$\|\|f(\mathbf{x})\|\|$		1.07 (1.07)
	Flops	6516	6492
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.9)	$\mathbf{x}$		
	$\|\|f(\mathbf{x})\|\|$		
	Flops	1521	1680
$\mathbf{x}_0 = [1 \quad 0.5]$ (P4.7.10)	$\mathbf{x}$	[0.2236 0.1273]	
	$\|\|f(\mathbf{x})\|\|$	0 (1.11e-16)	
	Flops	1278	2566

(cf) The numbers of floating-point operations and the residual (mismatching) errors in the parentheses are obtained from MATLAB 5.x version.

Table P4.8 Applying `newtons()` `fsolve()` for Systems of Nonlinear Equations

		`newtons()`	`fsolve()`
$\mathbf{x}_0 = [1 \quad 1 \quad 1]$	$\mathbf{x}$	[1.0000 -1.0000 1.0000]	
(P4.8.1)	$\|\|f(\mathbf{x})\|\|$	1.1102e-16 (1.1102e-16)	
	Flops	8158	12964
$\mathbf{x}_0 = [1 \quad 1 \quad 1]$	$\mathbf{x}$		[1 1 1]
(P4.8.2)	$\|\|f(\mathbf{x})\|\|$		0
	Flops	990	854
$\mathbf{x}_0 = [1 \quad 1 \quad 1]$	$\mathbf{x}$		
(P4.8.3)	$\|\|f(\mathbf{x})\|\|$		
	Flops	6611	4735
$\mathbf{x}_0 = [1 \quad 1 \quad 1]$	$\mathbf{x}$	[1.0000 -1.0000 1.0000]	
(P4.8.4)	$\|\|f(\mathbf{x})\|\|$	4.5506e-15 (4.6576e-15)	
	Flops	18,273	21,935
$\mathbf{x}_0 = [1 \quad 1 \quad 1]$	$\mathbf{x}$		
(P4.8.5)	$\|\|f(\mathbf{x})\|\|$		
	Flops	6811	5525
$\mathbf{x}_0 = [1 \quad 1 \quad 1]$	$\mathbf{x}$		[2.0000 1.0000 3.0000]
(P4.8.6)	$\|\|f(\mathbf{x})\|\|$		3.4659e-8 (2.6130e-8)
	Flops	6191	4884
$\mathbf{x}_0 = [1 \quad 1 \quad 1]$	$\mathbf{x}$	[1.0000 3.0000 2.0000]	
(P4.8.7)	$\|\|f(\mathbf{x})\|\|$	1.0022e-13 (1.0437e-13)	
	Flops	8055	6102

(e)
$$\begin{aligned} x^2 + y^2 + z^2 &= 14 \\ x^2 + 2y^2 - z &= 6 \\ x - 3y^2 + z^2 &= -2 \end{aligned} \tag{P4.8.5}$$

(f)
$$\begin{aligned} x^3 - 12y + z^2 &= 5 \\ 3x^2 + y^3 - 2z &= 7 \\ x + 24y^2 - 2\sin(\pi z/18) &= 25 \end{aligned} \tag{P4.8.6}$$

(g)
$$x^2 + y^2 - 2z = 6$$
$$x^2 - 2y + z^3 = 3 \tag{P4.8.7}$$
$$2xz - 3y^2 - z^2 = -27$$

4.9 Newton Method for a System of Nonlinear Equations with Varying Parameter(s)

In order to find the average modulation order x_i for each user of an OFDM (orthogonal frequency division multiplex) system that has $N(128)$ subchannels to assign to each of the four users in the environment of noise power N_0 and the bit error rate (probability of bit error) P_e, a communication system expert, Mi-hyun, formulated the problem into the system of five nonlinear equations as follows:

$$f_i(x) = (2^{x_i}(x_i \ln 2 - 1) + 1)\frac{N_0}{3} 2(\text{erfc}^{-1}(P_e/2))^2 - \lambda = 0 \tag{P4.9.1}$$

for $i = 1, 2, 3, 4$

$$f_5(x) = \sum_{i=1}^{4} \frac{a_i}{x_i} - N = 0 \tag{P4.9.2}$$

where $N = 128$ and a_i is the data rate of each user

where $\text{erfc}^{-1}(x)$ is the inverse function of the complementary error function

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt = 1 - \text{erf}(x) \tag{P4.9.3}$$

and defined as the MATLAB built-in function ‘`erfcinv()`’. She defined the mismatching error (vector) function as below and save it in the M-file named “`fp_bits.m`”.

```
function y = fp_bits(x,a,Pe)
%x(i),i = 1:4 correspond to the modulation order of each user
%x(5) corresponds to the Lagrange multiplier (Lambda)
if nargin < 3, Pe = 1e-4;
  if nargin < 2, a = [64 64 64 64]; end
end
N = 128; N0 = 1;
x14 = x(1:4);
y = (2.^x14.*(log(2)*x14 - 1)+1)*N0/3*2*erfcinv(Pe/2).^2 - x(5);
y(5) = sum(a./x14) - N;
```

Compose a program which solves the above system of nonlinear equations (with $N_0 = 1$ and $P_e = 10^{-4}$) to get the modulation order x_i of each user

for five different sets of data rates

a = [32 32 32 32], [64 32 32 32], [128 32 32 32], [256 32 32 32], and [512 32 32 32]

and plots a_1/x_1 (the number of subchannels assigned to user 1) versus a_1 (the data rate of user 1).

4.10 Temperature Rising from Heat Flux in a Semi-infinite Slab

Consider a semi-infinite slab whose temperature rises as a function of position $x > 0$ and time $t > 0$ as

$$T(x,t) = \frac{Qx}{k}\left(\frac{e^{-s^2}}{\sqrt{\pi}s} - \mathrm{erfc}(s)\right) \quad \text{with} \quad s^2 = x^2/4at \qquad \text{(P4.10.1)}$$

where the function erfc() is defined by Eq. (P4.9.3) and

$$Q \text{ (heat flux)} = 200 \text{ J/m}^2\text{s}, \quad k \text{ (conductivity)} = 0.015 \text{ J/m/s/}^\circ\text{C},$$
$$a \text{ (diffusivity)} = 2.5 \times 10^{-5} \text{ m}^2/\text{s}$$

In order to find the heat transfer speed, a heating system expert, Kyungwon, wants to solve the above equation to get the positions $x(t)$ with a temperature rise of $T = 30$ °C at $t =$ 10:10:200 s. Compose the program which does this job and plots $x(t)$ versus t.

4.11 Damped Newton Method for a Set of Nonlinear Equations

Consider the routine "`newtons()`", which is made for solving a system of equations and introduced in Section 4.6.

(a) Run the routine with the initial point $(x_{10}, x_{20}) = (0.5, 0.2)$ to solve Eq. (4.6.5) and certify that it does not yield the right solution as depicted in Fig. 4.6c.

(b) In order to keep the step size adjusted in the case where the norm of the vector function $\mathbf{f}(\mathbf{x}_{k+1})$ at iteration $k + 1$ is larger than that of $\mathbf{f}(\mathbf{x}_k)$ at iteration k, insert (activate) the statements numbered from 1 to 6 of the routine "`newtons()`" (Section 4.6) by deleting the comment mark (`%`) at the beginning of each line to make a modified routine "`newtonds()`", which implements the damped Newton method. Run it with the initial point $(x_{10}, x_{20}) = (0.5, 0.2)$ to solve Eq. (4.6.5) and certify that it yields the right solution as depicted in Fig. 4.6d.

(c) Run the MATLAB built-in routine "`fsolve()`" with the initial point $(x_{10}, x_{20}) = (0.5, 0.2)$ to solve Eq. (4.6.5). Does it present you a right solution?

5

NUMERICAL DIFFERENTIATION/ INTEGRATION

5.1 DIFFERENCE APPROXIMATION FOR FIRST DERIVATIVE

For a function $f(x)$ of a variable x, its first derivative is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{5.1.1}$$

However, this gives our computers a headache, since they do not know how to take a limit. Any input number given to computers must be a definite number and can be neither too small nor too large to be understood by the computer. The 'theoretically' infinitesimal number h involved in this equation is a problem.

A simple approximation that computers might be happy with is the forward difference approximation

$$D_{f1}(x, h) = \frac{f(x+h) - f(x)}{h} \qquad (h \text{ is step size}) \tag{5.1.2}$$

How far away is this approximation from the true value of (5.1.1)? In order to do the error analysis, we take the Taylor series expansion of $f(x+h)$ about x as

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f^{(2)}(x) + \frac{h^3}{3!} f^{(3)}(x) + \cdots \tag{5.1.3}$$

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

Subtracting $f(x)$ from both sides and dividing both sides by the step size h yields

$$D_{f1}(x,h) = \frac{f(x+h)-f(x)}{h} = f'(x) + \frac{h}{2}f^{(2)}(x) + \frac{h^2}{3!}f^{(3)}(x) + \cdots$$
$$= f'(x) + O(h) \tag{5.1.4}$$

where $O(g(h))$, called 'big Oh of $g(h)$', denotes a truncation error term proportional to $g(h)$ for $|h| \prec 1$. This means that the error of the forward difference approximation (5.1.2) of the first derivative is proportional to the step size h, or, equivalently, in the order of h.

Now, in order to derive another approximation formula for the first derivative having a smaller error, let's remove the first-order term with respect to h from Eq. (5.1.4) by substituting $2h$ for h in the equation

$$D_{f1}(x,2h) = \frac{f(x+2h)-f(x)}{2h} = f'(x) + \frac{2h}{2}f^{(2)}(x) + \frac{4h^2}{3!}f^{(3)}(x) + \cdots$$

and subtracting this result from two times the equation. Then, we get

$$2D_{f1}(x,h) - D_{f1}(x,2h) = 2\frac{f(x+h)-f(x)}{h} - \frac{f(x+2h)-f(x)}{2h}$$
$$= f'(x) - \frac{2h^2}{3!}f^{(3)}(x) + \cdots$$
$$D_{f2}(x,h) = \frac{2D_{f1}(x,h) - D_{f1}(x,2h)}{2-1}$$
$$= \frac{-f(x+2h)+4f(x+h)-3f(x)}{2h}$$
$$= f'(x) + O(h^2) \tag{5.1.5}$$

which can be regarded as an improvement over Eq. (5.1.4), since it has the truncation error of $O(h^2)$ for $|h| \prec 1$.

How about the backward difference approximation?

$$D_{b1}(x,h) = \frac{f(x)-f(x-h)}{h} \equiv D_{f1}(x,-h) \qquad (h \text{ is step size}) \tag{5.1.6}$$

This also has an error of $O(h)$ and can be processed to yield an improved version having a truncation error of $O(h^2)$.

$$D_{b2}(x,h) = \frac{2D_{b1}(x,h) - D_{b1}(x,2h)}{2-1} = \frac{3f(x)-4f(x-h)+f(x-2h)}{2h}$$
$$= f'(x) + O(h^2) \tag{5.1.7}$$

In order to derive another approximation formula for the first derivative, we take the Taylor series expansion of $f(x+h)$ and $f(x-h)$ up to the fifth order

to write

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f^{(2)}(x) + \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(4)}(x) + \frac{h^5}{5!}f^{(5)}(x) + \cdots$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f^{(2)}(x) - \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(4)}(x) - \frac{h^5}{5!}f^{(5)}(x) + \cdots$$

and divide the difference between these two equations by $2h$ to get the central difference approximation for the first derivative as

$$\begin{aligned} D_{c2}(x,h) = \frac{f(x+h) - f(x-h)}{2h} &= f'(x) + \frac{h^2}{3!}f^{(3)}(x) + \frac{h^4}{5!}f^{(5)}(x) + \cdots \\ &= f'(x) + O(h^2) \end{aligned} \tag{5.1.8}$$

which has an error of $O(h^2)$ similarly to Eqs. (5.1.5) and (5.1.7). This can also be processed to yield an improved version having a truncation error of $O(h^4)$.

$$\begin{aligned} 2^2 D_{c2}(x,h) - D_{c2}(x,2h) &= 4\frac{f(x+h) - f(x-h)}{2h} - \frac{f(x+2h) - f(x-2h)}{2 \cdot 2h} \\ &= 3f'(x) - \frac{12h^4}{5!}f^{(5)}(x) - \cdots \\ D_{c4}(x,h) &= \frac{2^2 D_{c1}(x,h) - D_{c1}(x,2h)}{2^2 - 1} \\ &= \frac{8f(x+h) - 8f(x-h) - f(x+2h) + f(x-2h)}{12h} \\ &= f'(x) + O(h^4) \end{aligned} \tag{5.1.9}$$

Furthermore, this procedure can be formularized into a general formula, called 'Richardson's extrapolation', for improving the difference approximation of the derivatives as follows:

<Richardson's extrapolation>

$$D_{f,n+1}(x,h) = \frac{2^n D_{f,n}(x,h) - D_{f,n}(x,2h)}{2^n - 1} \quad (n\text{: the order of error}) \tag{5.1.10a}$$

$$D_{b,n+1}(x,h) = \frac{2^n D_{b,n}(x,h) - D_{b,n}(x,2h)}{2^n - 1} \tag{5.1.10b}$$

$$D_{c,2(n+1)}(x,h) = \frac{2^{2n} D_{c,2n}(x,h) - D_{c,2n}(x,2h)}{2^{2n} - 1} \tag{5.1.10c}$$

5.2 APPROXIMATION ERROR OF FIRST DERIVATIVE

In the previous section, we derived some difference approximation formulas for the first derivative. Since their errors are proportional to some power of

the step-size h, it seems that the errors continue to decrease as h gets smaller. However, this is only half of the story since we considered only the truncation error caused by truncating the high-order terms in the Taylor series expansion and did not take account of the round-off error caused by quantization.

In this section, we will discuss the round-off error as well as the truncation error so as to gain a better understanding of how the computer really works. For this purpose, suppose that the function values

$$f(x+2h),\ f(x+h),\ f(x),\ f(x-h),\ f(x-2h)$$

are quantized (rounded-off) to

$$\begin{aligned} y_2 &= f(x+2h)+e_2, \qquad y_1 = f(x+h)+e_1 \\ y_0 &= f(x)+e_0 \\ y_{-1} &= f(x-h)+e_{-1}, \qquad y_{-2} = f(x-2h)+e_{-2} \end{aligned} \tag{5.2.1}$$

where the magnitudes of the round-off (quantization) errors e_2, e_1, e_0, e_{-1}, and e_{-2} are all smaller than some positive number ε, that is, $|e_i| \le \varepsilon$. Then, the total error of the forward difference approximation (5.1.4) can be derived as

$$D_{f1}(x,h) = \frac{y_1-y_0}{h} = \frac{f(x+h)+e_1-f(x)-e_0}{h} \overset{(5.1\text{-}4)}{=} f'(x) + \frac{e_1-e_0}{h} + \frac{K_1}{2}h$$

$$|D_{f1}(x,h)-f'(x)| \le \left|\frac{e_1-e_0}{h}\right| + \frac{|K_1|}{2}h \le \frac{2\varepsilon}{h} + \frac{|K_1|}{2}h \quad \text{with } K_1 = f^{(2)}(x)$$

Look at the right-hand side of this inequality—that is, the upper bound of error. It consists of two parts; the first one is due to the round-off error and in inverse proportion to the step-size h, while the second one is due to the truncation error and in direct proportion to h. Therefore, the upper bound of the total error can be minimized with respect to the step-size h to give the optimum step-size h_o as

$$\frac{d}{dh}\left(\frac{2\varepsilon}{h} + \frac{|K_1|}{2}h\right) = -\frac{2\varepsilon}{h^2} + \frac{|K_1|}{2} = 0, \qquad h_o = 2\sqrt{\frac{\varepsilon}{|K_1|}} \tag{5.2.2}$$

Thetotal error of the central difference approximation (5.1.8) can also be derived as follows:

$$\begin{aligned} D_{c2}(x,h) &= \frac{y_1-y_{-1}}{2h} = \frac{f(x+h)+e_1-f(x-h)-e_{-1}}{2h} \\ &\overset{(5.1.8)}{=} f'(x) + \frac{e_1-e_{-1}}{2h} + \frac{K_2}{6}h^2 \end{aligned}$$

$$|D_{c2}(x,h)-f'(x)| \le \left|\frac{e_1-e_{-1}}{2h}\right| + \frac{|K_1|}{6}h^2 \le \frac{2\varepsilon}{2h} + \frac{|K_2|}{6}h^2 \text{ with } K_2 = f^{(3)}(x)$$

The right-hand side of this inequality is minimized to yield the optimum step size h_o as

$$\frac{d}{dh}\left(\frac{\varepsilon}{h}+\frac{|K_2|}{6}h^2\right)=-\frac{\varepsilon}{h^2}+\frac{|K_2|}{3}h=0, \qquad h_o=\sqrt[3]{\frac{3\varepsilon}{|K_2|}} \tag{5.2.3}$$

Similarly, we can derive the total error of the central difference approximation (5.1.9) as

$$|D_{c4}(x,h)-f'(x)|\le\left|\frac{8e_1-8e_{-1}-e_2+e_{-2}}{12h}\right|+\frac{|K_4|}{30}h^4$$
$$\le\frac{18\varepsilon}{12h}+\frac{|K_4|}{30}h^4 \qquad \text{with } K_4=f^{(5)}(x)$$

and find out the optimum step size h_o as

$$\frac{d}{dh}\left(\frac{3\varepsilon}{2h}+\frac{|K_4|}{30}h^4\right)=-\frac{3\varepsilon}{2h^2}+\frac{2|K_4|}{15}h^3=0, \qquad h_o=\sqrt[5]{\frac{45\varepsilon}{4|K_4|}} \tag{5.2.4}$$

From what we have seen so far, we can tell that, as we make the step size h smaller, the round-off error may increase, while the truncation error decreases. This is called 'step-size dilemma'. Therefore, there must be some optimal step size h_o for the difference approximation formulas, as derived analytically in Eqs. (5.2.2), (5.2.3), and (5.2.4). However, these equations are only of theoretical value and cannot be used practically to determine h_o because we usually don't have any information about the high-order derivatives and, consequently, we cannot estimate $K_1, K_2, \ldots$. Besides, noting that h_o minimizes not the real error, but its upper bound, we can never expect the true optimal step size to be uniform for all x even with the same approximation formula.

Now, we can verify the step-size dilemma and the existence of some optimal step size h_o by computing the numerical derivative of a function, say, $f(x)=\sin x$, whose analytical derivatives are well known. To see how the errors of the difference approximation formulas (5.1.4) and (5.1.8) depend on the step size h, we computed their values for $x=\pi/4$ together with their errors as summarized in Tables 5.1 and 5.2. From these results, it appears that the errors of (5.1.4) and (5.1.8) are minimized with $h\approx 10^{-8}$ and $h\approx 10^{-5}$, respectively. This may be justified by the following facts:

- Noting that the number of significant bits is 52, which is the number of mantissa bits (Section 1.2.1), or, equivalently, the number of significant digits is about $52\times 3/10\approx 16$ (since $2^{10}\approx 10^3$), and the value of $f(x)=\sin x$ is less than or equal to one, the round-off error is roughly

$$\varepsilon\approx 10^{-16}/2$$

Table 5.1 The Forward Difference Approximation (5.1.4) for the First Derivative of $f(x) = \sin x$ and Its Error from the True Value ($\cos \pi/4 = 0.7071067812$) Depending on the Step Size h

$h_k = 10^{-k}$	$D_{1k\|x=\pi/4}$	$D_{1k} - D_{1(k-1)}$	$D_{1k\|x=\pi/4} - \cos(\pi/4)$
$h_1 = 0.1000000000$	0.6706029729		−0.03650380828
$h_2 = 0.0100000000$	0.7035594917	0.0329565188	−0.00354728950
$h_3 = 0.0010000000$	0.7067531100	0.0031936183	−0.00035367121
$h_4 = 0.0001000000$	0.7070714247	0.0003183147	−0.00003535652
$h_5 = 0.0000100000$	0.7071032456	0.0000318210	−0.00000353554
$h_6 = 0.0000010000$	0.7071064277	0.0000031821	−0.00000035344
$h_7 = 0.0000001000$	0.7071067454	0.0000003176	−0.0000003581
$h_8 = 0.0000000100$*	0.7071067842	0.0000000389	0.00000000305*
$h_9 = 0.0000000010$	0.7071068175	0.0000000333*	0.00000003636
$h_{10} = 0.0000000001$	0.7071077057	0.0000008882	0.00000092454
$h_o = 0.0000000168$ (the optimal value of h obtained from Eq. (5.2.2))			

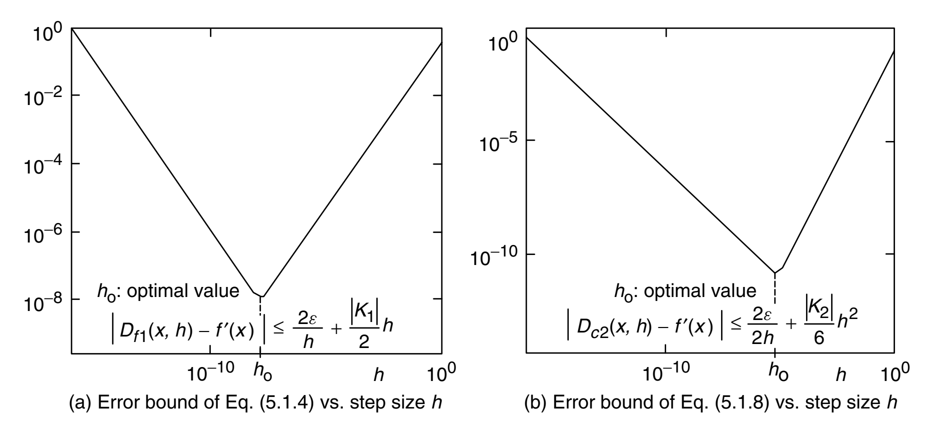
Table 5.2 The Forward Difference Approximation (5.1.8) for the First Derivative of $f(x) = \sin x$ and Its Error from the True Value ($\cos \pi/4 = 0.7071067812$) Depending on the Step Size h

$h_k = 10^{-k}$	$D_{2k\|x=\pi/4}$	$D_{2k} - D_{2(k-1)}$	$D_{2k\|x=\pi/4} - \cos(\pi/4)$
$h_1 = 0.1000000000$	0.7059288590		−0.00117792219
$h_2 = 0.0100000000$	0.7070949961	0.0011661371	−0.00001178505
$h_3 = 0.0010000000$	0.7071066633	0.0000116672	−0.00000011785
$h_4 = 0.0001000000$	0.7071067800	0.0000001167	−0.00000000118
$h_5 = 0.0000100000$*	0.7071067812	0.0000000012	−0.00000000001*
$h_6 = 0.0000010000$	0.7071067812	0.0000000001*	0.00000000005
$h_7 = 0.0000001000$	0.7071067804	−0.0000000009	−0.00000000084
$h_8 = 0.0000000100$	0.7071067842	0.0000000039	0.00000000305
$h_9 = 0.0000000010$	0.7071067620	−0.0000000222	−0.00000001915
$h_{10} = 0.0000000001$	0.7071071506	0.0000003886	0.00000036942
$h_o = 0.0000059640$ (the optimal value of h obtained from Eq. (5.2.3))			

- Accordingly, Eqs. (5.2.2) and (5.2.3) give the theoretical optimal values of step size h as

$$h_o = 2\sqrt{\frac{\varepsilon}{|K_1|}} = 2\sqrt{\frac{\varepsilon}{|f'(\pi/4)|}} = 2\sqrt{\frac{10^{-16}/2}{|-\sin(\pi/4)|}} = 1.68 \times 10^{-8}$$

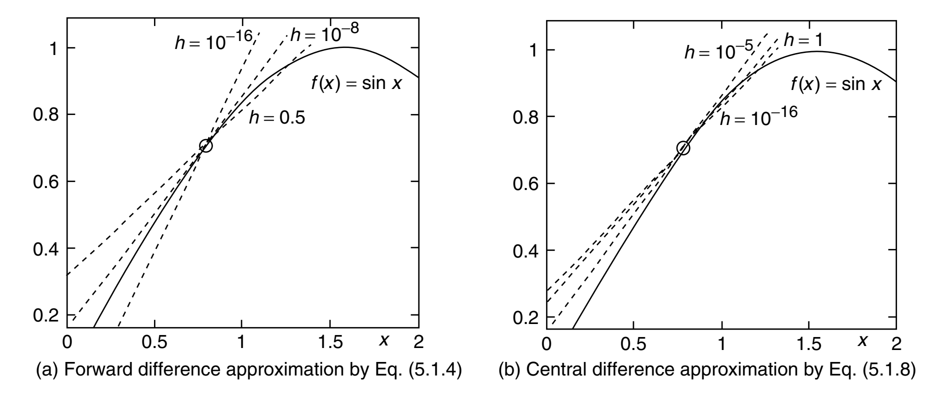
$$h_o = \sqrt[3]{\frac{3\varepsilon}{|K_2|}} = \sqrt[3]{\frac{3\varepsilon}{|f^{(3)}(\pi/4)|}} = \sqrt[3]{\frac{3 \times 10^{-16}/2}{|-\cos(\pi/4)|}} = 0.5964 \times 10^{-5}$$



(a) Error bound of Eq. (5.1.4) vs. step size h (b) Error bound of Eq. (5.1.8) vs. step size h

Figure 5.1 Forward/central difference approximation error of first derivative versus step size h.

Figure 5.1a/b shows how the error bounds of the difference approximations (5.1.4)/(5.1.8) for the first derivative vary with the step-size h, implying that there is some optimal value of step-size h with which the error bound of the numerical derivative is minimized. It seems that we might be able to get the optimal step-size h_o by using this kind of graph or directly using Eq. (5.2.2),(5.2.3) or (5.2.4). But, as mentioned before, it is not possible, as long as the high-order derivatives are unknown (as is usually the case). Very fortunately, Tables 5.1 and 5.2 suggest that we might be able to guess the good value of h by watching how small $|D_{ik} - D_{i(k-1)}|$ is for a given problem. On the other hand, Fig. 5.2a/b shows the tangential lines based on the forward/central difference approximations (5.1.4)/(5.1.8) of the first derivative at $x = \pi/4$ with the three values of step-size h. They imply that there is some optimal step-size h_o and the numerical approximation error becomes larger if we make the step-size h larger or smaller than the value.



(a) Forward difference approximation by Eq. (5.1.4) (b) Central difference approximation by Eq. (5.1.8)

Figure 5.2 Forward/central difference approximation of first derivative of $f(x) = \sin x$.

5.3 DIFFERENCE APPROXIMATION FOR SECOND AND HIGHER DERIVATIVE

In order to obtain an approximation formula for the second derivative, we take the Taylor series expansion of $f(x+h)$ and $f(x-h)$ up to the fifth order to write

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f^{(2)}(x) + \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(4)}(x) + \frac{h^5}{5!}f^{(5)}(x) + \cdots$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f^{(2)}(x) - \frac{h^3}{3!}f^{(3)}(x) + \frac{h^4}{4!}f^{(4)}(x) - \frac{h^5}{5!}f^{(5)}(x) + \cdots$$

Adding these two equations (to remove the $f'(x)$ terms) and then subtracting $2f(x)$ from both sides and dividing both sides by h^2 yields the central difference approximation for the second derivative as

$$\begin{aligned} D_{c2}^{(2)}(x,h) &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \\ &= f^{(2)}(x) + \frac{h^2}{12}f^{(4)}(x) + \frac{2h^4}{6!}f^{(6)}(x) + \cdots \end{aligned} \tag{5.3.1}$$

which has a truncation error of $O(h^2)$.

Richardson's extrapolation can be used for manipulating this equation to remove the h^2 term, which yields an improved version

$$\begin{aligned} \frac{2^2 D_{c2}^{(2)}(x,h) - D_{c2}^{(2)}(x,2h)}{2^2-1} &= \frac{-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)}{12h^2} \\ &= f^{(2)}(x) - \frac{h^4}{90}f^{(5)}(x) + \cdots \\ D_{c4}^{(2)}(x,h) &= \frac{-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)}{12h^2} \\ &= f^{(2)}(x) + O(h^4) \end{aligned} \tag{5.3.2}$$

which has a truncation error of $O(h^4)$.

The difference approximation formulas for the first and second derivatives derived so far are summarized in Table 5.3, where the following notations are used:

$D_{fi}^{(N)}/D_{bi}^{(N)}/D_{ci}^{(N)}$ is the forward/backward/central difference approximation for the Nth derivative having an error of $O(h^i)$(h is the step size)

$f_k = f(x+kh)$

Now, we turn our attention to the high-order derivatives. But, instead of deriving the specific formulas, let's make an algorithm to generate whatever difference approximation formula we want. For instance, if we want to get the approximation formula of the second derivative based on the function values f_2, f_1, f_0, f_{-1}, and f_{-2}, we write

$$D_{c4}^{(2)}(x,h) = \frac{c_2 f_2 + c_1 f_1 + c_0 f_0 + c_{-1} f_{-1} + c_{-2} f_{-2}}{h^2} \tag{5.3.3}$$

and take the Taylor series expansion of f_2, f_1, f_{-1}, and f_{-2} excluding f_0 on the right-hand side of this equation to rewrite it as

$$D_{c4}^{(2)}(x,h)$$
$$= \frac{1}{h^2}\left\{\begin{array}{l} c_2\left(f_0 + 2hf_0' + \dfrac{(2h)^2}{2}f_0^{(2)} + \dfrac{(2h)^3}{3!}f_0^{(3)} + \dfrac{(2h)^4}{4!}f_0^{(4)} + \cdots\right) \\ +c_1\left(f_0 + hf_0' + \dfrac{h^2}{2}f_0^{(2)} + \dfrac{h^3}{3!}f_0^{(3)} + \dfrac{h^4}{4!}f_0^{(4)} + \cdots\right) + c_0 f_0 \\ +c_{-1}\left(f_0 - hf_0' + \dfrac{h^2}{2}f_0^{(2)} - \dfrac{h^3}{3!}f_0^{(3)} + \dfrac{h^4}{4!}f_0^{(4)} - \cdots\right) \\ +c_{-2}\left(f_0 - 2hf_0' + \dfrac{(2h)^2}{2}f_0^{(2)} - \dfrac{(2h)^3}{3!}f_0^{(3)} + \dfrac{(2h)^4}{4!}f_0^{(4)} - \cdots\right) \end{array}\right\}$$

$$= \frac{1}{h^2}\left\{\begin{array}{c} (c_2 + c_1 + c_0 + c_{-1} + c_{-2})f_0 + h(2c_2 + c_1 - c_{-1} - 2c_{-2})f_0' \\ +h^2\left(\dfrac{2^2}{2}c_2 + \dfrac{1}{2}c_1 + \dfrac{1}{2}c_{-1} + \dfrac{2^2}{2}c_{-2}\right)f_0^{(2)} \\ +h^3\left(\dfrac{2^3}{3!}c_2 + \dfrac{1}{3!}c_1 - \dfrac{1}{3!}c_{-1} - \dfrac{2^3}{3!}c_{-2}\right)f_0^{(3)} \\ +h^4\left(\dfrac{2^4}{4!}c_2 + \dfrac{1}{4!}c_1 + \dfrac{1}{4!}c_{-1} + \dfrac{2^4}{4!}c_{-2}\right)f_0^{(4)} + \cdots \end{array}\right\} \tag{5.3.4}$$

We should solve the following set of equations to determine the coefficients c_2, c_1, c_0, c_{-1}, and c_{-2} so as to make the expression conform to the second derivative $f_0^{(2)}$ at $x + 0h = x$.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 0 & -1 & -2 \\ 2^2/2! & 1/2! & 0 & 1/2! & 2^2/2! \\ 2^3/3! & 1/3! & 0 & -1/3! & -2^3/3! \\ 2^4/4! & 1/4! & 0 & 1/4! & 2^4/4! \end{bmatrix} \begin{bmatrix} c_2 \\ c_1 \\ c_0 \\ c_{-1} \\ c_{-2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{5.3.5}$$

Table 5.3 The Difference Approximation Formulas for the First and Second Derivatives

$O(h)$ forward difference approximation for the first derivative:

$$D_{f1}(x,h) = \frac{f_1 - f_0}{h} \tag{5.1.4}$$

$O(h^2)$ forward difference approximation for the first derivative:

$$D_{f2}(x,h) = \frac{2D_{f1}(x,h) - D_{f1}(x,2h)}{2-1} = \frac{-f_2 + 4f_1 - 3f_0}{2h} \tag{5.1.5}$$

$O(h)$ backward difference approximation for the first derivative:

$$D_{b1}(x,h) = \frac{f_0 - f_{-1}}{h} \tag{5.1.6}$$

$O(h^2)$ backward difference approximation for the first derivative:

$$D_{b2}(x,h) = \frac{2D_{b1}(x,h) - D_{b1}(x,2h)}{2-1} = \frac{3f_0 - 4f_{-1} + f_{-2}}{2h} \tag{5.1.7}$$

$O(h^2)$ central difference approximation for the first derivative:

$$D_{c2}(x,h) = \frac{f_1 - f_{-1}}{2h} \tag{5.1.8}$$

$O(h^4)$ forward difference approximation for the first derivative:

$$D_{c4}(x,h) = \frac{2^2 D_{c2}(x,h) - D_{c2}(x,2h)}{2^2-1} = \frac{-f_2 + 8f_1 - 8f_{-1} + f_{-2}}{12h} \tag{5.1.9}$$

$O(h^2)$ central difference approximation for the second derivative:

$$D_{c2}^{(2)}(x,h) = \frac{f_1 - 2f_0 + f_{-1}}{h^2} \tag{5.3.1}$$

$O(h^4)$ forward difference approximation for the second derivative:

$$D_{c4}^{(2)}(x,h) = \frac{2^2 D_{c2}^{(2)}(x,h) - D_{c2}^{(2)}(x,2h)}{2^2-1} = \frac{-f_2 + 16f_1 - 30f_0 + 16f_{-1} - f_{-2}}{12h^2} \tag{5.3.2}$$

$O(h^2)$ central difference approximation for the fourth derivative:

$$D_{c2}^{(4)}(x,h) = \frac{f_{-2} - 4f_{-1} + 6f_0 - 4f_1 + f_2}{h^4} \text{(from difapx(4,[-2 2])} \tag{5.3.6}$$

```
function [c,err,eoh,A,b] = difapx(N,points)
%difapx.m to get the difference approximation for the Nth derivative
l = max(points);
L = abs(points(1)-points(2))+ 1;
if L < N + 1, error('More points are needed!'); end
for n = 1: L
   A(1,n) = 1;
   for m = 2:L + 2, A(m,n) = A(m - 1,n)*l/(m - 1); end %Eq.(5.3.5)
   l = l-1;
end
b = zeros(L,1); b(N + 1) = 1;
c =(A(1:L,:)\b)'; %coefficients of difference approximation formula
err = A(L + 1,:)*c'; eoh = L-N; %coefficient & order of error term
if abs(err) < eps, err = A(L + 2,:)*c'; eoh = L - N + 1; end
if points(1) < points(2), c = fliplr(c); end
```

The procedure of setting up this equation and solving it is cast into the MATLAB routine "`difapx()`", which can be used to generate the coefficients of, say, the approximation formulas (5.1.7), (5.1.9), and (5.3.2) just for practice/verification/fun, whatever your purpose is.

```
>>format rat %to make all numbers represented in rational form
>>difapx(1,[0 -2]) %1st derivative based on {f0, f-1, f-2}
   ans = 3/2    -2    1/2 %Eq.(5.1-7)
>>difapx(1,[-2  2]) %1st derivative based on {f-2, f-1, f0, f1, f2}
   ans = 1/12  -2/3    0    2/3  -1/12 %Eq.(5.1.9)
>>difapx(2,[2 -2]) %2nd derivative based on {f2, f1, f0, f-1, f-2}
   ans = -1/12  4/3  -5/2   4/3  -1/12 %Eq.(5.3.2)
```

Example 5.1. Numerical/Symbolic Differentiation for Taylor Series Expansion. Consider how to use MATLAB to get the Taylor series expansion of a function—say, e^{-x} about $x = 0$—which we already know is

$$e^{-x} = 1 - x + \frac{1}{2}x^2 - \frac{1}{3!}x^3 + \frac{1}{4!}x^4 - \frac{1}{5!}x^5 + \cdots \tag{E5.1.1}$$

As a numerical method, we can use the MATLAB routine "`difapx()`". On the other hand, we can also use the MATLAB command "`taylor()`", which is a symbolic approach. Readers may put '`help taylor`' into the MATLAB command window to see its usage, which is restated below.

- `taylor(f)` gives the fifth-order Maclaurin series expansion of `f`.
- `taylor(f,n + 1)` with an integer `n` $>$ 0 gives the nth-order Maclaurin series expansion of `f`.
- `taylor(f,a)` with a real number(`a`) gives the fifth-order Taylor series expansion of `f` about `a`.

- `taylor(f,n + 1,a)` gives the n th-order Taylor series expansion of `f` about default_variable = `a`.
- `taylor(f,n + 1,a,y)` gives the nth-order Taylor series expansion of `f(y)` about `y = a`.

(cf) The target function `f` must be a legitimate expression given directly as the first input argument.

(cf) Before using the command "`taylor()`", one should declare the arguments of the function as symbols by putting the statement like "`syms x t`".

(cf) In the case where the function has several arguments, it is a good practice to put the independent variable as the last input argument of "`taylor()`", though `taylor()` takes one closest (alphabetically) to '`x`' as the independent variable by default only if it has been declared as a symbolic variable and is contained as an input argument of the function `f`.

(cf) One should use the MATLAB command "`sym2poly()`" if he wants to extract the coefficients from the Taylor series expansion obtained as a symbolic expression.

The following MATLAB program "`nm5e01`" finds us the coefficients of fifth-order Taylor series expansion of e^{-x} about $x = 0$ by using the two methods.

```
%nm5e01:Nth-order Taylor series expansion for e^-x about xo in Ex 5.1
f=inline('exp(-x)','x');
N = 5; xo = 0;
%Numerical computation method
T(1) = feval(f,xo);
h = 0.005 %.01 or 0.001 make it worse
tmp = 1;
for i = 1:N
   tmp = tmp*i*h; %i!(factorial i)*h^i
   c = difapx(i,[-i i]); %coefficient of numerical derivative
   dix = c*feval(f,xo + [-i:i]*h)'; %/h^i; %derivative
   T(i+1) = dix/tmp; %Taylor series coefficient
end
format rat, Tn = fliplr(T) %descending order
%Symbolic computation method
syms x; Ts = sym2poly(taylor(exp(-x),N + 1,xo))
%discrepancy
format short, discrepancy=norm(Tn - Ts)
```

5.4 INTERPOLATING POLYNOMIAL AND NUMERICAL DIFFERENTIAL

The difference approximation formulas derived in the previous sections are applicable only when the target function $f(x)$ to differentiate is somehow given. In this section, we think about how to get the numerical derivatives when we are

given only the data file containing several data points. A possible measure is to make the interpolating function by using one of the methods explained in Chapter 3 and get the derivative of the interpolating function.

For simplicity, let's reconsider the problem of finding the derivative of $f(x) = \sin x$ at $x = \pi/4$, where the function is given as one of the following data point sets:

$$\left\{\left(\frac{\pi}{8}, \sin\frac{\pi}{8}\right), \left(\frac{\pi}{4}, \sin\frac{\pi}{4}\right), \left(\frac{3\pi}{8}, \sin\frac{3\pi}{8}\right)\right\}$$

$$\left\{(0, \sin 0), \left(\frac{\pi}{8}, \sin\frac{\pi}{8}\right), \left(\frac{\pi}{4}, \sin\frac{\pi}{4}\right), \left(\frac{3\pi}{8}, \sin\frac{3\pi}{8}\right), \left(\frac{4\pi}{8}, \sin\frac{4\pi}{8}\right)\right\}$$

$$\left\{\left(\frac{2\pi}{16}, \sin\frac{2\pi}{16}\right), \left(\frac{3\pi}{16}, \sin\frac{3\pi}{16}\right), \left(\frac{4\pi}{16}, \sin\frac{4\pi}{16}\right), \left(\frac{5\pi}{16}, \sin\frac{5\pi}{16}\right), \left(\frac{6\pi}{16}, \sin\frac{6\pi}{16}\right)\right\}$$

We make the MATLAB program "`nm540`", which uses the routine "`lagranp()`" to find the interpolating polynomial, uses the routine "`polyder()`" to differentiate the polynomial, and computes the error of the resulting derivative from the true value. Let's run it with `x` defined appropriately according to the given set of data points and see the results.

```
>>nm540
  dfx( 0.78540) = 0.689072 (error: -0.018035) %with x = [1:3]*pi/8
  dfx( 0.78540) = 0.706556 (error: -0.000550) %with x = [0:4]*pi/8
  dfx( 0.78540) = 0.707072 (error: -0.000035) %with x = [2:6]*pi/16
```

This illustrates that if we have more points that are distributed closer to the target point, we may get better result.

```
%nm540
% to interpolate by Lagrange polynomial and get the derivative
clear, clf
x0 = pi/4;
df0 = cos(x0); % True value of derivative of sin(x) at x0 = pi/4
for m = 1:3
  if m == 1, x = [1:3]*pi/8;
   elseif  m == 2, x = [0:4]*pi/8;
   else   x = [2:6]*pi/16;
  end
  y = sin(x);
  px = lagranp(x,y); % Lagrange polynomial interpolating (x,y)
  dpx = polyder(px); % derivative of polynomial px
  dfx = polyval(dpx, x0);
  fprintf(' dfx(%6.4f) = %10.6f (error: %10.6f)\n', x0,dfx,dfx - df0);
end
```

One more thing to mention before closing this section is that we have the MATLAB built-in routine "`diff()`", which finds us the difference vector for a given vector. When the data points $\{(x_k, f(x_k)), k = 1, 2, \ldots\}$ are given as an

ASCII data file named "`xy.dat`", we can use the routine "`diff()`" to get the divided difference, which is similar to the derivative of a continuous function.

```
>>load xy.dat %input the contents of 'xy.dat' as a matrix named xy
>>dydx = diff(xy(:,2))./diff(xy(:,1)); dydx' %divided difference
  dydx =  2.0000   0.50000   2.0000
```

k	x_k `xy(:,1)`	$f(x_k)$ `xy(:,2)`	$x_{k+1} - x_k$ `diff(xy(:,1))`	$f(x_{k+1}) - f(x_k)$ `diff(xy(:,2))`	$D_k = \dfrac{f(x_{k+1}) - f(x_k)}{x_{k+1} - x_k}$
1	−1	2	1	2	2
2	0	4	2	1	1/2
3	2	5	−1	−2	2
4	1	3			

5.5 NUMERICAL INTEGRATION AND QUADRATURE

The general form of numerical integration of a function $f(x)$ over some interval $[a, b]$ is a weighted sum of the function values at a finite number $(N + 1)$ of sample points (nodes), referred to as 'quadrature':

$$\int_a^b f(x)\,dx \cong \sum_{k=0}^{N} w_k f(x_k) \qquad \text{with } a = x_0 < x_1 < \cdots < x_N = b \qquad (5.5.1)$$

Here, the sample points are equally spaced for the midpoint rule, the trapezoidal rule, and Simpson's rule, while they are chosen to be zeros of certain polynomials for Gaussian quadrature.

Figure 5.3 shows the integrations over two segments by the midpoint rule, the trapezoidal rule, and Simpson's rule, which are referred to as Newton–Cotes formulas for being based on the approximate polynomial and are implemented by the following formulas.

$$\langle\text{midpoint rule}\rangle \qquad \int_{x_k}^{x_{k+1}} f(x)\,dx \cong h f_{mk} \qquad (5.5.2)$$

$$\text{with } h = x_{k+1} - x_k, \quad f_{mk} = f(x_{mk}),\ x_{mk} = \frac{x_k + x_{k+1}}{2}$$

$$\langle\text{trapezoidal rule}\rangle \qquad \int_{x_k}^{x_{k+1}} f(x)\,dx \cong \frac{h}{2}(f_k + f_{k+1}) \qquad (5.5.3)$$

$$\text{with } h = x_{k+1} - x_k, \quad f_k = f(x_k)$$

$$\langle\text{Simpson's rule}\rangle \qquad \int_{x_{k-1}}^{x_{k+1}} f(x)\,dx \cong \frac{h}{3}(f_{k-1} + 4f_k + f_{k+1}) \qquad (5.5.4)$$

$$\text{with } h = \frac{x_{k+1} - x_{k-1}}{2}$$

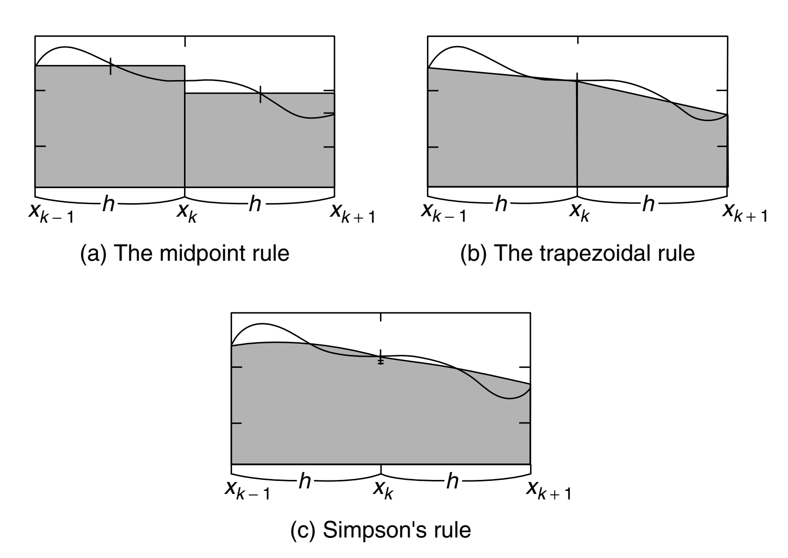


Figure 5.3 Various methods of numerical integration.

These three integration rules are based on approximating the target function (integrand) to the zeroth-, first- and second-degree polynomial, respectively. Since the first two integrations are obvious, we are going to derive just Simpson's rule (5.5.4). For simplicity, we shift the graph of $f(x)$ by $-x_k$ along the x axis, or, equivalently, make the variable substitution $t = x - x_k$ so that the abscissas of the three points on the curve of $f(x)$ change from $x = \{x_k - h, x_k, x_k + h\}$ to $t = \{-h, 0, +h\}$. Then, in order to find the coefficients of the second-degree polynomial

$$p_2(t) = c_1 t^2 + c_2 t + c_3 \tag{5.5.5}$$

matching the points $(-h, f_{k-1})$, $(0, f_k)$, $(+h, f_{k+1})$, we should solve the following set of equations:

$$\begin{aligned} p_2(-h) &= c_1(-h)^2 + c_2(-h) + c_3 = f_{k-1} \\ p_2(0) &= c_1 0^2 + c_2 0 + c_3 \qquad\quad\;\; = f_k \\ p_2(+h) &= c_1(+h)^2 + c_2(+h) + c_3 = f_{k+1} \end{aligned}$$

to determine the coefficients c_1, c_2, and c_3 as

$$c_3 = f_k, \qquad c_2 = \frac{f_{k+1} - f_{k-1}}{2h}, \qquad c_1 = \frac{1}{h^2}\left(\frac{f_{k+1} + f_{k-1}}{2} - f_k\right)$$

Integrating the second-degree polynomial (5.5.5) with these coefficients from $t = -h$ to $t = h$ yields

$$\int_{-h}^{h} p_2(t)\,dt = \frac{1}{3}c_1t^3 + \frac{1}{2}c_2t^2 + c_3t\Big|_{-h}^{h} = \frac{2}{3}c_1h^3 + 2c_3h$$
$$= \frac{2h}{3}\left(\frac{f_{k+1}+f_{k-1}}{2} - f_k + 3f_k\right) = \frac{h}{3}(f_{k-1} + 4f_k + f_{k+1})$$

This is the Simpson integration formula (5.5.4).

Now, as a preliminary work toward diagnosing the errors of the above integration formulas, we take the Taylor series expansion of the integral function

$$g(x) = \int_{x_k}^{x} f(t)\,dt \quad \text{with } g'(x) = f(x),\ \ g^{(2)}(x) = f'(x),\ \ g^{(3)}(x) = f^{(2)}(x) \tag{5.5.6}$$

about the lower bound x_k of the integration interval to write

$$g(x) = g(x_k) + g'(x_k)(x - x_k) + \frac{1}{2}g^{(2)}(x_k)(x - x_k)^2 + \frac{1}{3!}g^{(3)}(x_k)(x - x_k)^3 + \cdots$$

Substituting Eq. (5.5.6) together with $x = x_{k+1}$ and $x_{k+1} - x_k = h$ into this yields

$$\int_{x_k}^{x_{k+1}} f(x)\,dx = 0 + hf(x_k) + \frac{h^2}{2}f'(x_k) + \frac{h^3}{3!}f^{(2)}(x_k) + \frac{h^4}{4!}f^{(3)}(x_k) + \frac{h^5}{5!}f^{(4)}(x_k) + \cdots \tag{5.5.7}$$

First, for the error analysis of the midpoint rule, we substitute x_{k-1} and $-h = x_{k-1} - x_k$ in place of x_{k+1} and h in this equation to write

$$\int_{x_k}^{x_{k-1}} f(x)\,dx = 0 - hf(x_k) + \frac{h^2}{2}f'(x_k) - \frac{h^3}{3!}f^{(2)}(x_k) + \frac{h^4}{4!}f^{(3)}(x_k) - \frac{h^5}{5!}f^{(4)}(x_k) + \cdots$$

and subtract this equation from Eq. (5.5.7) to write

$$\int_{x_k}^{x_{k+1}} f(x)\,dx - \int_{x_k}^{x_{k-1}} f(x)\,dx = \int_{x_k}^{x_{k+1}} f(x)\,dx + \int_{x_{k-1}}^{x_k} f(x)\,dx$$
$$= \int_{x_{k-1}}^{x_{k+1}} f(x)\,dx = 2hf(x_k) + \frac{2h^3}{3!}f^{(2)}(x_k) + \frac{2h^5}{5!}f^{(4)}(x_k) + \cdots \tag{5.5.8}$$

Substituting x_k and $x_{mk} = (x_k + x_{k+1})/2$ in place of x_{k-1} and x_k in this equation and noting that $x_{k+1} - x_{mk} = x_{mk} - x_k = h/2$, we obtain

$$\int_{x_k}^{x_{k+1}} f(x)\,dx = hf(x_{mk}) + \frac{h^3}{3\times 2^3}f^{(2)}(x_{mk}) + \frac{h^5}{5\times 4\times 3\times 2^5}f^{(4)}(x_{mk}) + \cdots$$

$$\int_{x_k}^{x_{k+1}} f(x)\,dx - hf(x_{mk}) = \frac{h^3}{24}f^{(2)}(x_{mk}) + \frac{h^5}{1920}f^{(4)}(x_{mk}) + \cdots = O(h^3) \tag{5.5.9}$$

This, together with Eq. (5.5.2), implies that the error of integration over one segment by the midpoint rule is proportional to h^3.

Second, for the error analysis of the trapezoidal rule, we subtract Eq. (5.5.3) from Eq. (5.5.7) to write

$$\begin{aligned}
&\int_{x_k}^{x_{k+1}} f(x)\,dx - \frac{h}{2}(f(x_k) + f(x_{k+1})) \\
&= hf(x_k) + \frac{h^2}{2}f'(x_k) + \frac{h^3}{3!}f^{(2)}(x_k) + \frac{h^4}{4!}f^{(3)}(x_k) + \frac{h^5}{5!}f^{(4)}(x_k) + \cdots \\
&\quad - \frac{h}{2}\left(f(x_k) + f(x_k) + hf'(x_k) + \frac{h^2}{2}f^{(2)}(x_k) + \frac{h^3}{3!}f^{(3)}(x_k)\right. \\
&\qquad \left. + \frac{h^4}{4!}f^{(4)}(x_k) + \cdots\right) \\
&= -\frac{h^3}{12}f^{(2)}(x_k) - \frac{h^4}{24}f^{(3)}(x_k) - \frac{h^5}{80}f^{(4)}(x_k) + O(h^6) = O(h^3) \qquad (5.5.10)
\end{aligned}$$

This implies that the error of integration over one segment by the trapezoidal rule is proportional to h^3.

Third, for the error analysis of Simpson's rule, we subtract the Taylor series expansion of Eq. (5.5.4)

$$\begin{aligned}
&\frac{h}{3}(f(x_{k-1}) + 4f(x_k) + f(x_{k+1})) \\
&= \frac{h}{3}\left(f(x_k) + 4f(x_k) + f(x_k) + \frac{2h^2}{2}f^{(2)}(x_k) + \frac{2h^4}{4!}f^{(4)}(x_k) + \cdots\right) \\
&= 2hf(x_k) + \frac{h^3}{3}f^{(2)}(x_k) + \frac{h^5}{36}f^{(4)}(x_k) + \cdots
\end{aligned}$$

from Eq. (5.5.8) to write

$$\begin{aligned}
\int_{x_{k-1}}^{x_{k+1}} f(x)\,dx - \frac{h}{3}(f(x_{k-1}) + 4f(x_k) + f(x_{k+1})) &= -\frac{h^5}{90}f^{(4)}(x_k) + O(h^7) \\
&= O(h^5) \qquad (5.5.11)
\end{aligned}$$

This implies that the error of integration over two segments by Simpson's rule is proportional to h^5.

Before closing this section, let's make use of these error equations to find a way of estimating the error of the numerical integral from the true integral without knowing the derivatives of the target (integrand) function $f(x)$. For this purpose, we investigate how the error of numerical integration by Simpson's rule

$$I_S(x_{k-1}, x_{k+1}, h) = \frac{h}{3}(f(x_{k-1}) + 4f(x_k) + f(x_{k+1}))$$

will change if the segment width h is halved to $h/2$. Noting that, from Eq. (5.5.11),

$$E_S(h) = \int_{x_{k-1}}^{x_{k+1}} f(x)\,dx - I_S(x_{k-1}, x_{k+1}, h) \approx -\frac{h^5}{90} f^{(4)}(c)(c \in [x_{k-1}, x_{k+1}])$$

$$\begin{aligned} E_S(\frac{h}{2}) &= \int_{x_{k-1}}^{x_{k+1}} f(x)\,dx - I_S\left(x_{k-1}, x_{k+1}, \frac{h}{2}\right) \\ &= \int_{x_{k-1}}^{x_k} f(x)\,dx - I_S\left(x_{k-1}, x_k, \frac{h}{2}\right) + \int_{x_k}^{x_{k+1}} f(x)\,dx \\ &\quad - I_S\left(x_k, x_{k+1}, \frac{h}{2}\right) (c \in [x_{k-1}, x_{k+1}]) \\ &\approx -2\frac{(h/2)^5}{90} f^{(4)}(c) = \frac{1}{16} E_S(h) \end{aligned}$$

we can express the change of the error caused by halving the segment width as

$$\begin{aligned} \left|E_S(h) - E_S\left(\frac{h}{2}\right)\right| &= \left|I_S(x_{k-1}, x_{k+1}, h) - I_S\left(x_{k-1}, x_{k+1}, \frac{h}{2}\right)\right| \\ &\approx \frac{15}{16}|E_S(h)| \approx 15\left|E_S\left(\frac{h}{2}\right)\right| \end{aligned} \tag{5.5.12}$$

This suggests the error estimate of numerical integration by Simpson's rule as

$$\left|E_S\left(\frac{h}{2}\right)\right| \approx \frac{1}{2^4 - 1}\left|I_S(x_{k-1}, x_{k+1}, h) - I_S\left(x_{k-1}, x_{k+1}, \frac{h}{2}\right)\right| \tag{5.5.13}$$

Also for the trapezoidal rule, similar result can be derived:

$$\left|E_T\left(\frac{h}{2}\right)\right| \approx \frac{1}{2^2 - 1}\left|I_T(x_{k-1}, x_{k+1}, h) - I_T\left(x_{k-1}, x_{k+1}, \frac{h}{2}\right)\right| \tag{5.5.14}$$

5.6 TRAPEZOIDAL METHOD AND SIMPSON METHOD

In order to get the formulas for numerical integration of a function $f(x)$ over some interval $[a, b]$, we divide the interval into N segments of equal length $h = (b - a)/N$ so that the nodes (sample points) can be expressed as $\{x = a + kh, k = 0, 1, 2, \ldots, N\}$. Then we have the numerical integration of $f(x)$ over $[a, b]$ by the trapezoidal rule (5.5.3) as

$$\begin{aligned} \int_a^b f(x)\,dx &= \sum_{k=0}^{N-1} \int_{x_k}^{x_{k+1}} f(x)\,dx \\ &\cong \frac{h}{2}\{(f_0 + f_1) + (f_1 + f_2) + \cdots + (f_{N-2} + f_{N-1}) + (f_{N-1} + f_N)\} \end{aligned}$$

$$I_{T2}(a,b,h) = h\left\{\frac{f(a)+f(b)}{2} + \sum_{k=1}^{N-1} f(x_k)\right\} \tag{5.6.1}$$

whose error is proportional to h^2 as N times the error for one segment [Eq. (5.5.10)], that is,

$$NO(h^3) = (b-a)/h \times O(h^3) = O(h^2)$$

On the other hand, we have the numerical integration of $f(x)$ over $[a,b]$ by Simpson's rule (5.5.4) with an even number of segments N as

$$\int_a^b f(x)\,dx = \sum_{m=0}^{N/2-1}\int_{x_{2m}}^{x_{2m+2}} f(x)\,dx$$

$$\cong \frac{h}{3}\{(f_0+4f_1+f_2)+(f_2+4f_3+f_4)+\cdots+(f_{N-2}+4f_{N-1}+f_N)\}$$

$$I_{S4}(a,b,h) = \frac{h}{3}\left\{f(a)+f(b)+4\sum_{m=0}^{N/2-1} f(x_{2m+1}) + 2\sum_{m=1}^{N/2-1} f(x_{2m})\right\} \tag{5.6.2}$$

$$= \frac{h}{3}\left\{f(a)+f(b)+2\left(\sum_{m=0}^{N/2-1} f(x_{2m+1}) + \sum_{k=1}^{N-1} f(x_k)\right)\right\}$$

whose error is proportional to h^4 as N times the error for one segment [Eq. (5.5.11)], that is,

$$(N/2)O(h^5) = (b-a)/2h \times O(h^5) = O(h^4)$$

These two integration formulas by the trapezoidal rule and Simpson's rule are cast into the MATLAB routines "`trpzds()`" and "`smpsns()`", respectively.

```
function INTf = trpzds(f,a,b,N)
%integral of f(x) over [a,b] by trapezoidal rule with N segments
if abs(b - a) < eps | N <= 0, INTf = 0; return; end
h = (b - a)/N; x = a +[0:N]*h; fx = feval(f,x); values of f for all nodes
INTf = h*((fx(1) + fx(N + 1))/2 + sum(fx(2:N))); %Eq.(5.6.1)
```

```
function INTf = smpsns(f,a,b,N,varargin)
%integral of f(x) over [a,b] by Simpson's rule with N segments
if nargin < 4, N = 100; end
if abs(b - a)<1e-12 | N <= 0, INTf = 0; return; end
if mod(N,2) ~= 0, N = N + 1; end  %make N even
h = (b - a)/N; x = a + [0:N]*h; %the boundary nodes for N segments
fx = fevel(f,x,varargin{:}); %values of f for all nodes
fx(find(fx == inf)) = realmax; fx(find(fx == -inf)) = -realmax;
kodd = 2:2:N; keven = 3:2:N - 1; %the set of odd/even indices
INTf = h/3*(fx(1) + fx(N + 1)+4*sum(fx(kodd)) + 2*sum(fx(keven)));%Eq.(5.6.2)
```

5.7 RECURSIVE RULE AND ROMBERG INTEGRATION

In this section, we are going to look for a recursive formula which enables us to use some numerical integration with the segment width h to produce another (hopefully better) numerical integration with half the segment width ($h/2$). Additionally, we use Richardson extrapolation (Section 5.1) together with the two successive numerical integrations to make a Romberg table that can be used to improve the accuracy of the numerical integral step by step.

Let's start with halving the segment width h to $h/2$ for the trapezoidal method. Then, the numerical integration formula (5.6.1) can be written in the recursive form as

$$\begin{aligned} I_{T2}\left(a, b, \frac{h}{2}\right) &= \frac{h}{2}\left\{\frac{f(a)+f(b)}{2} + \sum_{k=1}^{2N-1} f(x_{k/2})\right\} \\ &= \frac{h}{2}\left\{\frac{f(a)+f(b)}{2} + \sum_{m=1}^{N-1} f(x_{2m/2}) + \sum_{m=0}^{N-1} f(x_{(2m+1)/2})\right\} \\ &= \frac{1}{2}\left\{I_{T2}(a, b, h) + \sum_{m=0}^{N-1} f(x_{(2m+1)/2})(\text{terms for inserted nodes})\right\} \end{aligned} \tag{5.7.1}$$

Noting that the error of this formula is proportional to h^2 ($O(h^2)$), we apply a Richardson extrapolation [Eq. (5.1.10)] to write a higher-level integration formula having an error of $O(h^4)$ as

$$\begin{aligned} I_{T4}(a, b, h) &= \frac{2^2 I_{T2}(a, b, h) - I_{T2}(a, b, 2h)}{2^2 - 1} \\ &\overset{(5.6.1)}{=} \frac{1}{3}\left\{4\frac{h}{2}\left(f(a)+f(b)+2\sum_{k=1}^{N-1} f(x_k)\right)\right. \\ &\qquad \left. -\frac{2h}{2}\left(f(a)+f(b)+2\sum_{m=1}^{N/2-1} f(x_{2m})\right)\right\} \\ &= \frac{h}{3}\left\{f(a)+f(b)+4\sum_{m=1}^{N/2} f(x_{2m-1}) + 2\sum_{m=1}^{N/2-1} f(x_{2m})\right\} \\ &\overset{(5.6.2)}{\equiv} I_{S4}(a, b, h) \end{aligned} \tag{5.7.2}$$

which coincides with the Simpson's integration formula. This implies that we don't have to distinguish the trapezoidal rule from Simpson's rule. Anyway,

replacing h by $h/2$ in this equation yields

$$I_{T4}\left(a, b, \frac{h}{2}\right) = \frac{2^2 I_{T2}(a, b, h/2) - I_{T2}(a, b, h)}{2^2 - 1}$$

which can be generalized to the following formula:

$$I_{T,2(n+1)}(a, b, 2^{-(k+1)}h) = \frac{2^{2n} I_{T,2n}(a, b, 2^{-(k+1)}h) - I_{T,2n}(a, b, 2^{-k}h)}{2^{2n} - 1} \quad \text{for } n \geq 1, k \geq 0 \tag{5.7.3}$$

Now, it is time to introduce a systematic way, called Romberg integration, of improving the accuracy of the integral step by step and estimating the (truncation) error at each step to determine when to stop. It is implemented by a Romberg Table (Table 5.4), that is, a lower-triangular matrix that we construct one row per iteration by applying Eq. (5.7.1) in halving the segment width h to get the next-row element (downward in the first column), and applying Eq. (5.7.3) in upgrading the order of error to get the next-column elements (rightward in the row) based on the up–left (north–west) one and the left (west) one. At each iteration k, we use Eq. (5.5.14) to estimate the truncation error as

$$\left|E_{T,2(k+1)}(2^{-k}h)\right| \approx \frac{1}{2^{2k} - 1}\left|I_{T,2k}(2^{-k}h) - I_{T,2k}(2^{-(k-1)}h)\right| \tag{5.7.4}$$

and stop the iteration when the estimated error becomes less than some prescribed tolerance. Then, the last diagonal element is taken to be 'supposedly' the best

```
function [x,R,err,N] = rmbrg(f,a,b,tol,K)
%construct Romberg table to find definite integral of f over [a,b]
h = b - a; N = 1;
if nargin < 5, K = 10; end
R(1,1) = h/2*(feval(f,a)+ feval(f,b));
for k = 2:K
   h = h/2;  N = N*2;
   R(k,1) = R(k - 1,1)/2 + h*sum(feval(f,a +[1:2:N - 1]*h)); %Eq.(5.7.1)
   tmp = 1;
   for n = 2:k
      tmp = tmp*4;
      R(k,n) = (tmp*R(k,n - 1)-R(k - 1,n - 1))/(tmp - 1); %Eq.(5.7.3)
   end
   err = abs(R(k,k - 1)- R(k - 1,k - 1))/(tmp - 1); %Eq.(5.7.4)
   if err < tol,  break;  end
end
x = R(k,k);
```

estimate of the integral. This sequential procedure of Romberg integration is cast into the MATLAB routine "`rmbrg()`".

Before closing this section, we test and compare the trapezoidal method ("`trpzds()`"), Simpson method ("`smpsns()`"), and Romberg integration ("`rmbrg()`") by trying them on the following integral

Table 5.4 Romberg Table

Iteration k	Segment Width h	$n = 1$	$n = 2$	$n = 3$	......
0	h_0	$I_{T,2}(h_0)$			
		(5.7.1)↓ (5.7.3)↘			
1	$2^{-1}h_0$	$I_{T,2}(2^{-1}h_0) \longrightarrow$	$I_{T,4}(2^{-1}h_0)$		
		(5.7.1)↓ (5.7.3)↘	(5.7.3)↘		
2	$2^{-2}h_0$	$I_{T,2}(2^{-2}h_0) \longrightarrow$	$I_{T,4}(2^{-2}h_0) \longrightarrow$	$I_{T,6}(2^{-2}h_0)$	
•	•	•	•	•	•

$$\int_0^4 400x(1-x)e^{-2x}\,dx = 100\left\{-2e^{-2x}x(1-x)\Big|_0^4 + \int_0^4 2e^{-2x}(1-2x)\,dx\right\}$$

$$= 100\left\{-2e^{-2x}x(1-x)\Big|_0^4 - e^{-2x}(1-2x)\Big|_0^4 - 2\int_0^4 e^{-2x}\,dx\right\}$$

$$= 200x^2e^{-2x}\Big|_0^4 = 3200e^{-8} = 1.07348040929 \tag{5.7.5}$$

Here are the MATLAB statements for this job listed together with the running results.

```
>>f = inline('400*x.*(1 - x).*exp(-2*x)','x');
>>a = 0; b = 4; N = 80;
>>format short e
>>true_I = 3200*exp(-8)
>>It = trpzds(f,a,b,N), errt = It-true_I %trapezoidal
   It = 9.9071e-001, errt = -8.2775e-002

>>Is = smpsns(f,a,b,N), errs = Is-true_I %Simpson
   INTfs = 1.0731e+000, error = -3.3223e-004

>>[IR,R,err,N1] = rmbrg(f,a,b,.0005), errR = IR - true_I %Romberg
   INTfr = 1.0734e+000, N1 = 32
   error = -3.4943e-005
```

As expected from the fact that the errors of numerical integration by the trapezoidal method and Simpson method are $O(h^2)$ and $O(h^4)$, respectively, the Simpson method presents better results (with smaller error) than the trapezoidal

one with the same number of segments $N = 80$. Moreover, Romberg integration with $N = 32$ shows a better result than both of them.

5.8 ADAPTIVE QUADRATURE

The numerical integration methods in the previous sections divide the integration interval uniformly into the segments of equal width, making the error nonuniform over the interval—that is, small/large for smooth/swaying portion of the curve of integrand $f(x)$. In contrast, the strategy of the adaptive quadrature is to divide the integration interval nonuniformly into segments of (generally) unequal lengths—that is, short/long segments for swaying/smooth portion of the curve of integrand $f(x)$, aiming at having smaller error with fewer segments.

The algorithm of adaptive quadrature scheme starts with a numerical integral (`INTf`) for the whole interval and the sum of numerical integrals (`INTf12 = INTf1 + INTf2`) for the two segments of equal width. Based on the difference between the two successive estimates `INTf` and `INTf12`, it estimates the error of `INTf12` by using Eq. (5.5.13)/(5.5.14) depending on the basic integration rule. Then, if the error estimate is within a given tolerance (`tol`), it terminates with `INTf12`. Otherwise, it digs into each segment by repeating the same procedure with half of the tolerance (`tol/2`) assigned to both segments, until the deepest level satisfies the error condition. This is how the adaptive scheme forms sections of nonuniform width, as illustrated in Fig. 5.4. In fact, this algorithm really fits the nested (recursive) calling structure introduced in Section 1.3 and is cast into

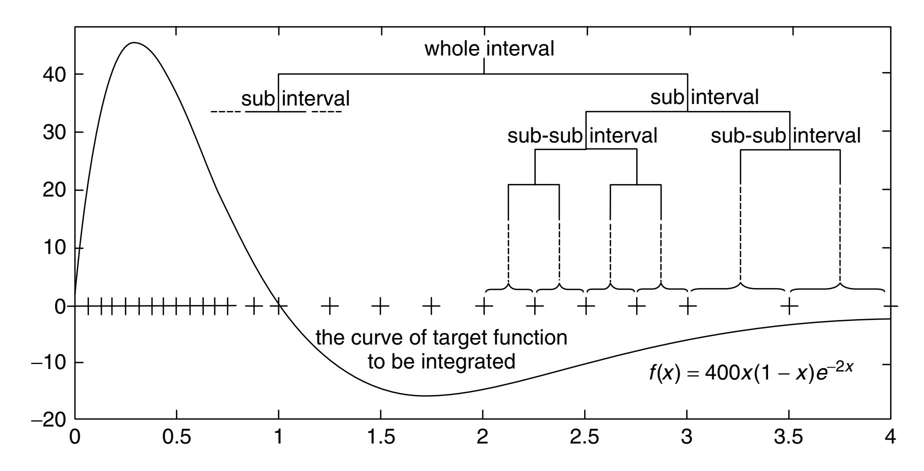


Figure 5.4 The subintervals (segments) and their boundary points (nodes) determined by the adaptive Simpson method.

the routine "adap_smpsn()", which needs the calling routine "adapt_smpsn()" for start-up.

```
function [INTf,nodes,err] = adap_smpsn(f,a,b,INTf,tol,varargin)
%adaptive recursive Simpson method
c = (a+b)/2;
INTf1 = smpsns(f,a,c,1,varargin{:});
INTf2 = smpsns(f,c,b,1,varargin{:});
INTf12 = INTf1 + INTf2;
err = abs(INTf12 - INTf)/15; % Error estimate by Eq.(5.5.13)
if isnan(err) | err < tol | tol<eps % NaN? Satisfying error? Too deep level?
  INTf = INTf12;
  points = [a c b];
 else
  [INTf1,nodes1,err1] = adap_smpsn(f,a,c,INTf1,tol/2,varargin{:});
  [INTf2,nodes2,err2] = adap_smpsn(f,c,b,INTf2,tol/2,varargin{:});
  INTf = INTf1 + INTf2;
  nodes = [nodes1 nodes2(2:length(nodes2))];
  err = err1 + err2;
end
```

```
function [INTf,nodes,err] = adapt_smpsn(f,a,b,tol,varargin)
%apply adaptive recursive Simpson method
INTf = smpsns(f,a,b,1,varargin{:});
[INTf,nodes,err] = adap_smpsn(f,a,b,INTf,tol,varargin{:});
```

We can apply these routines to get the approximate value of integration (5.7.5) by putting the following MATLAB statements into the MATLAB command window.

```
>>f = inline('400*x.*(1 - x).*exp(-2*x)','x');
>>a=0; b = 4; tol = 0.001;
>>format short e
>>true_I = 3200*exp(-8);
>>Ias = adapt_smpsn(f,a,b,tol), erras=Ias-true_I
   Ias = 1.0735e+000, erras = -8.9983e-006
```

Figure 5.4 shows the curve of the integrand $f(x) = 400x(1 - x)e^{-2x}$ together with the 25 nodes determined by the routine "adapt_smpsn()", which yields better results (having smaller error) with fewer segments than other methods discussed so far. From this figure, we see that the nodes are dense/sparse in the swaying/smooth portion of the curve of the integrand.

Here, we introduce the MATLAB built-in routines adopting the adaptive recursive integration scheme together with the illustrative example of their usage.

```
"quad(f,a,b,tol,trace,p1,p2,..)" / "quadl(f,a,b,tol,trace,p1,p2,..)"
```

```
>>Iq = quad(f,a,b,tol), errq = Iq - true_I
   Iq = 1.0735e+000, errq = 4.0107e-005
>>Iql = quadl(f,a,b,tol), errql = Iql - true_I
   Iql = 1.0735e+000, errq1 = -1.2168e-008
```

(cf) These routines are capable of passing the parameters (`p1,p2,..`) to the integrand (target) function and can be asked to show a list of intermediate subintervals with the fifth input argument `trace=1`.

(cf) `quadl()` is introduced in MATLAB 6.x version to replace another adaptive integration routine `quad8()` which is available in MATLAB 5.x version.

Additionally, note that MATLAB has a symbolic integration routine "`int(f,a,b)`". Readers may type "`help int`" into the MATLAB command window to see its usage, which is restated below.

- `int(f)` gives the indefinite integral of `f` with respect to its independent variable (closest to '`x`').
- `int(f,v)` gives the indefinite integral of `f(v)` with respect to `v` given as the second input argument.
- `int(f,a,b)` gives the definite integral of `f` over [`a,b`] with respect to its independent variable.
- `int(f,v,a,b)` gives the definite integral of `f(v)` with respect to `v` over [`a,b`].

(cf) The target function `f` must be a legitimate expression given directly as the first input argument and the upper/lower bound `a,b` of the integration interval can be a symbolic scalar or a numeric.

Example 5.2. Numerical/Symbolic Integration using `quad()/quadl()/int()`.

Consider how to make use of MATLAB for obtaining the continuous-time Fourier series (CtFS) coefficient

$$X_k = \int_{-P/2}^{P/2} x(t)e^{-jk\omega_0 t}\,dt = \int_{-P/2}^{P/2} x(t)e^{-j2\pi kt/P}\,dt \tag{E5.2.1}$$

For simplicity, let's try to get just the 16th CtFS coefficient of a rectangular wave

$$x(t) = \begin{cases} 1 & \text{for } -1 \le t < 1 \\ 0 & \text{for } -2 \le t < 1 \text{ or } 1 \le t < 2 \end{cases} \tag{E5.2.2}$$

which is periodic in t with period $P = 4$. We can compute it analytically as

$$\begin{aligned} X_{16} &= \int_{-2}^{2} x(t)e^{-j2\pi 16t/4}\,dt = \int_{-1}^{1} e^{-j8\pi t}\,dt = \frac{1}{-j8\pi}e^{-j8\pi t}\Big|_{-1}^{1} \\ &= \frac{1}{8\pi}\sin(8\pi t)\Big|_{-1}^{1} = 0 \end{aligned} \tag{E5.2.3}$$

```
%nm5e02
%use quad()/quad8() and int() to get CtFS coefficient X16 in Ex 5.2
ftn = 'exp(-j*k*w0*t)';  fcos = inline(ftn,'t','k','w0');
P = 4; k = 16; w0 = 2*pi/P;
a = -1; b = 1; tol = 0.001; trace = 0;
X16_quad = quad(fcos,a,b,tol,trace,k,w0)
X16_quadl = quadl(fcos,a,b,tol,trace,k,w0)
syms t; % declare symbolic variable
Iexp = int(exp(-j*k*w0*t),t) % symbolic indefinite integral
Icos = int(cos(k*w0*t),t) % symbolic indefinite integral
X16_sym = int(cos(k*w0*t),t,-1,1) % symbolic definite integral
```

As a numerical approach, we can use the MATLAB routine "quad()"/ "quadl()". On the other hand, we can also use the MATLAB routine "int()", which is a symbolic approach. We put all the statements together to make the MATLAB program "nm5e02", in which the fifth input argument (trace) of "quad()"/"quadl()" is set to 1 so that we can see their nodes and tell how different they are. Let's run it and see the results.

```
>>nm5e02
   X16_quad = 0.8150 + 0.0000i %betrayal of MATLAB?
   X16_quadl = 7.4771e-008 %almost zero, OK!
   Iexp = 1/8*i/pi*exp(-8*i*pi*t) %(E5.2.3) by symbolic computation
   Icos = 1/8/pi*sin(8*pi*t) %(E5.2.3) by symbolic computation
   X16_sym = 0 %exact answer by symbolic computation
```

What a surprise! It is totally unexpected that the MATLAB routine "quad()" gives us a quite eccentric value (0.8150), even without any warning message. The routine "quad()" must be branded as a betrayer for a piecewise-linear function multiplied by a periodic function. This seems to imply that "quadl()" is better than "quad()" and that "int()" is the best of the three commands. It should, however, be noted that "int()" can directly accept and handle only the functions composed of basic mathematical functions, rejecting the functions defined in the form of string or by the "inline()" command or through an m-file and besides, it takes a long time to execute.

(cf) What about our lovely routine "adapt_smpsn()"? Regrettably, you had better not count on it, since it will give the wrong answer for this problem. Actually, "quadl()" is much more reliable than "quad()" and "adapt_smpsn()".

5.9 GAUSS QUADRATURE

In this section, we cover several kinds of Gauss quadrature methods—that is, Gauss–Legendre integration, Gauss–Hermite integration, Gauss–Laguerre integration and Gauss–Chebyshev I,II integration. Each tries to approximate one of

the following integrations, respectively:

$$\int_a^b f(t)\,dt, \qquad \int_{-\infty}^{+\infty} e^{-t^2} f(t)\,dt, \qquad \int_0^{+\infty} e^{-t} f(t)\,dt,$$

$$\int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} f(t)\,dt, \quad \int_{-1}^{1} \sqrt{1-t^2} f(t)dt \quad \approx \quad \sum_{i=1}^{N} w_i f(t_i)$$

The problem is how to fix the weight w_i's and the (Gauss) grid points t_i's.

5.9.1 Gauss–Legendre Integration

If the integrand $f(t)$ is a polynomial of degree $\leq 3(= 2N - 1)$, then its integration

$$I(-1, 1) = \int_{-1}^{+1} f(t)\,dt \tag{5.9.1}$$

can exactly be obtained from just 2(N) points by using the following formula

$$I[t_1, t_2] = w_1 f(t_1) + w_2 f(t_2) \tag{5.9.2}$$

How marvelous it is! It is almost a magic. Do you doubt it? Then, let's find the weights w_1, w_2 and the grid points t_1, t_2 such that the approximating formula (5.9.2) equals the integration (5.9.1) for $f(t) = 1$(of degree 0), t(of degree 1), t^2(of degree 2), and t^3(of degree 3). In order to do so, we should solve the following system of equations:

$$f(t) = 1: \qquad w_1 f(t_1) + w_2 f(t_2) = w_1 + w_2 \equiv \int_{-1}^{1} 1\,dt = 2 \tag{5.9.3a}$$

$$f(t) = t: \qquad w_1 f(t_1) + w_2 f(t_2) = w_1 t_1 + w_2 t_2 \equiv \int_{-1}^{1} t\,dt = 0 \tag{5.9.3b}$$

$$f(t) = t^2: \qquad w_1 f(t_1) + w_2 f(t_2) = w_1 t_1^2 + w_2 t_2^2 \equiv \int_{-1}^{1} t^2\,dt = \frac{2}{3} \tag{5.9.3c}$$

$$f(t) = t^3: \qquad w_1 f(t_1) + w_2 f(t_2) = w_1 t_1^3 + w_2 t_2^3 \equiv \int_{-1}^{1} t^3\,dt = 0 \tag{5.9.3d}$$

Multiplying (5.9.3b) by t_1^2 and subtracting the result from (5.9.3d) yields

$$w_2(t_2^3 - t_1^2 t_2) = w_2 t_2 (t_2 + t_1)(t_2 - t_1) = 0 \rightarrow t_2 = -t_1, \quad t_2 = t_1 \text{(meaningless)}$$

$$t_2 = -t_1 \rightarrow \text{(5.9.3b)}, \qquad (w_1 - w_2) t_1 = 0,$$

$$w_1 = w_2 \rightarrow \text{(5.9.3a)}, \qquad w_1 + w_1 = 2$$

$$w_1 = w_2 = 1 \rightarrow \text{(5.9.3c)}, \quad t_1^2 + (-t_1)^2 = \frac{2}{3}, \quad t_1 = -t_2 = -\frac{1}{\sqrt{3}}$$

so that Eq. (5.9.2) becomes

$$I[t_1, t_2] = f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \tag{5.9.4}$$

We can expect this approximating formula to give us the exact value of the integral (5.9.1) when the integrand $f(t)$ is a polynomial of degree ≤ 3.

Now, you are concerned about how to generalize this two-point Gauss–Legendre integration formula to an N-point case, since a system of nonlinear equation like Eq. (5.9.3) can be very difficult to solve as the dimension increases. But, don't worry about it. The N grid points (t_i's) of Gauss–Legendre integration formula

$$I_{GL}[t_1, t_2, \ldots, t_N] = \sum_{i=1}^{N} w_{N,i} f(t_i) \tag{5.9.5}$$

giving us the exact integral of an integrand polynomial of degree $\le (2N - 1)$ can be obtained as the zeros of the Nth-degree Legendre polynomial [K-1, Section 4.3]

$$L_N(t) = \sum_{i=0}^{\lfloor N/2 \rfloor} (-1)^i \frac{(2N-2i)!}{2^N i!(N-i)!(N-2i)!} t^{N-2i} \tag{5.9.6a}$$

$$L_N(t) = \frac{1}{N}((2N-1)tL_{N-1}(t) - (N-1)L_{N-2}(t)) \tag{5.9.6b}$$

Given the N grid point t_i's, we can get the corresponding weight $w_{N,i}$'s of the N-point Gauss–Legendre integration formula by solving the system of linear equations

$$\begin{bmatrix} 1 & 1 & 1 & \bullet & 1 \\ t_1 & t_2 & t_n & \bullet & t_N \\ t_1^{n-1} & t_2^{n-1} & t_n^{n-1} & \bullet & t_N^{n-1} \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ t_1^{N-1} & t_2^{N-1} & t_n^{N-1} & \bullet & t_N^{N-1} \end{bmatrix} \begin{bmatrix} w_{N,1} \\ w_{N,2} \\ w_{N,n} \\ \bullet \\ w_{N,N} \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ (1-(-1)^n)/n \\ \bullet \\ (1-(-1)^N)/N \end{bmatrix} \tag{5.9.7}$$

where the nth element of the right-hand side (RHS) vector is

$$\mathrm{RHS}(n) = \int_{-1}^{1} t^{n-1}\,dt = \frac{1}{n}t^n \Big|_{-1}^{1} = \frac{1-(-1)^n}{n} \tag{5.9.8}$$

This procedure of finding the N grid point t_i's and the weight $w_{N,i}$'s of the N-point Gauss–Legendre integration formula is cast into the MATLAB routine "`Gausslp()`". We can get the two grid point t_i's and the weight $w_{N,i}$'s of the two-point Gauss–Legendre integration formula by just putting the following statement into the MATLAB command window.

```
function [t,w] = Gausslp(N)
if N < 0, fprintf('\nGauss-Legendre polynomial of negative order??\n');
 else
   t = roots(Lgndrp(N))'; %make it a row vector
   A(1,:) = ones(1,N);  b(1) = 2;
   for n = 2:N % Eq.(5.9.7)
      A(n,:) = A(n - 1,:).*t;
      if mod(n,2) == 0, b(n) = 0;
       else  b(n) = 2/n; % Eq.(5.9.8)
      end
   end
   w = b/A';
end
```

```
function p = Lgndrp(N) %Legendre polynomial
if N <= 0, p = 1; %n*Ln(t) = (2n - 1)t Ln - 1(t)-(n - 1)Ln-2(t) Eq.(5.9.6b)
 elseif N == 1, p = [1 0];
 else p = ((2*N - 1)*[Lgndrp(N - 1) 0]-(N - 1)*[0 0 Lgndrp(N - 2)])/N;
end
```

```
function I = Gauss_Legendre(f,a,b,N,varargin)

%Gauss_Legendre integration of f over [a,b] with N grid points
% Never try N larger than 25
[t,w] = Gausslp(N);
x = ((b - a)*t + a + b)/2; %Eq.(5.9.9)
fx = feval(f,x,varargin{:});
I = w*fx'*(b - a)/2; %Eq.(5.9.10)
```

```
>>[t,w] = Gausslp(2)
    t =    0.5774    -0.5774          w =    1          1
```

Even though we are happy with the N-point Gauss–Legendre integration formula (5.9.1) giving the exact integral of polynomials of degree $\le (2N - 1)$, we do not feel comfortable with the fixed integration interval $[-1, +1]$. But, we can be relieved from the stress because any arbitrary finite interval $[a, b]$ can be transformed into $[-1, +1]$ by the variable substitution known as the Gauss–Legendre translation

$$x = \frac{(b-a)t + a + b}{2}, \qquad dx = \frac{b-a}{2}\,dt \tag{5.9.9}$$

Then, we can write the N-point Gauss–Legendre integration formula for the integration interval $[a, b]$ as

$$I[a,b] = \int_a^b f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f(x(t))\,dt$$

$$I[x_1, x_2, \ldots, x_N] = \frac{b-a}{2}\sum_{i=1}^{N} w_{N,i}\, f(x_i) \quad \text{with } x_i = \frac{(b-a)t_i + a + b}{2} \tag{5.9.10}$$

The scheme of integrating $f(x)$ over the interval $[a, b]$ by the N-point Gauss–Legendre formula is cast into the MATLAB routine “`Gauss_Legendre()`”. We

can get the integral (5.7.5) by simply putting the following statements into the MATLAB command window. The result shows that the 10-point Gauss–Legendre formula yields better accuracy (smaller error), even with fewer nodes/segments than other methods discussed so far.

```
>>f = inline('400*x.*(1 - x).*exp(-2*x)','x'); %Eq.(5.7.5)
>>format short e
>>true_I = 3200*exp(-8);
>>a = 0; b = 4; N = 10; %integration interval & number of nodes(grid points)
>>IGL = gauss_legendre(f,a,b,N), errGL = IGL-true_I
   IGL = 1.0735e+000, errGL = 1.6289e-009
```

5.9.2 Gauss–Hermite Integration

The Gauss–Hermite integration formula is expressed by Eq. (5.9.5) as

$$I_{GH}[t_1, t_2, \ldots, t_N] = \sum_{i=1}^{N} w_{N,i} f(t_i) \tag{5.9.11}$$

and is supposed to give us the exact integral of the exponential e^{-t^2} multiplied by a polynomial $f(t)$ of degree $\leq (2N - 1)$ over $(-\infty, +\infty)$

$$I = \int_{-\infty}^{+\infty} e^{-t^2} f(t)\,dt \tag{5.9.12}$$

The N grid point t_i's can be obtained as the zeros of the N-point Hermite polynomial [K-1, Section 4.8]

$$H_N(t) = \sum_{i=0}^{\lfloor N/2 \rfloor} \frac{(-1)^i}{i!} N(N-1)\cdots(N-2i+1)(2t)^{N-2i} \tag{5.9.13a}$$

$$H_N(t) = 2t H_{N-1}(t) - H'(t) \tag{5.9.13b}$$

```
function [t,w] = Gausshp(N)
if N < 0
  error('Gauss-Hermite polynomial of negative degree??');
end
t = roots(Hermitp(N))';
A(1,:) = ones(1,N);  b(1) = sqrt(pi);
for n = 2:N
   A(n,:) = A(n - 1,:).*t; %Eq.(5.9.7)
   if mod(n,2) == 1, b(n) = (n - 2)/2*b(n - 2); %Eq.(5.9.14)
     else   b(n) = 0;
   end
end
w = b/A';
```

```
function p = Hermitp(N)
%Hn + 1(x) = 2xHn(x)-Hn'(x) from 'Advanced Engineering Math' by Kreyszig
if N <= 0,  p = 1;
 else  p = [2 0];
       for n = 2:N, p = 2*[p 0]-[0 0 polyder(p)]; end %Eq.(5.9.13b)
end
```

Given the N grid point t_i's, we can get the weight $w_{N,i}$'s of the N-point Gauss–Hermite integration formula by solving the system of linear equations like Eq. (5.9.7), but with the right-hand side (RHS) vector as

$$\begin{aligned}\text{RHS}(1) &= \int_{-\infty}^{\infty} e^{-t^2}\,dt = \sqrt{\int_{-\infty}^{\infty} e^{-x^2}\,dx \int_{-\infty}^{\infty} e^{-y^2}\,dy} \\ &= \sqrt{\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-(x^2+y^2)}\,dx\,dy} = \sqrt{\int_{-\infty}^{\infty} e^{-r^2} 2\pi\ r\,dr} \\ &= \sqrt{-\pi e^{-r^2}\Big|_0^{\infty}} = \sqrt{\pi} \end{aligned} \tag{5.9.14a}$$

$$\begin{aligned}\text{RHS}(n) &= \int_{-\infty}^{\infty} e^{-t^2} t^{n-1}\,dt = \int_{-\infty}^{\infty} (-2t)e^{-t^2}\frac{1}{-2}t^{n-2}\,dt\ \ (= 0 \text{ if } n \text{ is even}) \\ &= -\frac{1}{2}e^{-t^2}t^{n-2}\Big|_{-\infty}^{\infty} + \frac{1}{2}(n-2)\int_{-\infty}^{\infty} e^{-t^2}t^{n-3}\,dt = \frac{1}{2}(n-2)\text{RHS}(n-2)\end{aligned} \tag{5.9.14b}$$

The procedure for finding the N grid point t_i's and the corresponding weight $w_{N,i}$'s of the N-point Gauss–Hermite integration formula is cast into the MATLAB routine "`Gausshp()`". Note that, even though the integrand function ($g(t)$) doesn't have e^{-t^2} as a multiplying factor, we can multiply it by $e^{-t^2}e^{t^2} = 1$ to fabricate it as if it were like in Eq. (5.9.12):

$$I = \int_{-\infty}^{\infty} g(t)\,dt = \int_{-\infty}^{\infty} e^{-t^2}(e^{t^2}g(t))\,dt = \int_{-\infty}^{\infty} e^{-t^2} f(t)\,dt \tag{5.9.15}$$

5.9.3 Gauss–Laguerre Integration

The Gauss–Laguerre integration formula is also expressed by Eq. (5.9.5) as

$$I_{GLa}[t_1, t_2, \ldots, t_N] = \sum_{i=1}^{N} w_{N,i} f(t_i) \tag{5.9.16}$$

and is supposed to give us the exact integral of the exponential e^{-t} multiplied by a polynomial $f(t)$ of degree $\le (2N-1)$ over $[0, \infty)$

$$I = \int_0^{\infty} e^{-t} f(t)\,dt \tag{5.9.17}$$

The N grid point t_i's can be obtained as the zeros of the Nth-degree Laguerre polynomial [K-1, Section 4.7]

$$L_N(t) = \sum_{i=0}^{N} \frac{(-1)^i}{i!}\frac{N^i}{(N-i)!\ i!}t^i \tag{5.9.18}$$

Given the N grid point t_i's, we can get the corresponding weight $w_{N,i}$'s of the N-point Gauss–Laguerre integration formula by solving the system of linear equations like Eq. (5.9.7), but with the right-hand side (RHS) vector as

$$\text{RHS}(1) = \int_0^\infty e^{-t}\,dt = -e^{-t}\Big|_0^\infty = 1 \tag{5.9.19a}$$

$$\begin{aligned}\text{RHS}(n) &= \int_0^\infty e^{-t}t^{n-1}\,dt = -e^{-t}t^{n-1}\Big|_0^\infty + (n-1)\int_0^\infty e^{-t}t^{n-2}\,dt \\ &= (n-1)\text{RHS}(n-1)\end{aligned} \tag{5.9.19b}$$

5.9.4 Gauss–Chebyshev Integration

The Gauss–Chebyshev I integration formula is also expressed by Eq. (5.9.5) as

$$I_{GC1}[t_1, t_2, \ldots, t_N] = \sum_{i=1}^{N} w_{N,i} f(t_i) \tag{5.9.20}$$

and is supposed to give us the exact integral of $1/\sqrt{1-t^2}$ multiplied by a polynomial $f(t)$ of degree $\le (2N-1)$ over $[-1, +1]$

$$I = \int_{-1}^{+1} \frac{1}{\sqrt{1-t^2}} f(t)\,dt \tag{5.9.21}$$

The N grid point t_i's are the zeros of the Nth-degree Chebyshev polynomial (Section 3.3)

$$t_i = \cos\frac{(2i-1)\pi}{2N} \qquad \text{for } i = 1, 2, \ldots, N \tag{5.9.22}$$

and the corresponding weight $w_{N,i}$'s are uniformly selected as

$$w_{N,i} = \pi/N, \qquad \forall\, i = 1, \ldots, N \tag{5.9.23}$$

The Gauss–Chebyshev II integration formula is also expressed by Eq. (5.9.5) as

$$I_{GC2}[t_1, t_2, \ldots, t_N] = \sum_{i=1}^{N} w_{N,i} f(t_i) \tag{5.9.24}$$

and is supposed to give us the exact integral of $\sqrt{1-t^2}$ multiplied by a polynomial $f(t)$ of degree $\le (2N-1)$ over $[-1, +1]$

$$I = \int_{-1}^{+1} \sqrt{1-t^2}\, f(t)\,dt \tag{5.9.25}$$

The N grid point t_i's and the corresponding weight $w_{N,i}$'s are

$$t_i = \cos\left(\frac{i\pi}{N+1}\right), \qquad w_{N,i} = \frac{\pi}{N+1}\sin^2\left(\frac{i\pi}{N+1}\right) \quad \text{for } i = 1, 2, \ldots, N \tag{5.9.26}$$

5.10 DOUBLE INTEGRAL

In this section, we consider the numerical integration of a function $f(x, y)$ with respect to two variables x and y over the integration region $R = \{(x, y)|a \le x \le b, c(x) \le y \le d(x)\}$ as depicted in Fig. 5.5.

$$I = \iint_R f(x, y)\,dx\,dy = \int_a^b \left\{\int_{c(x)}^{d(x)} f(x, y)dy\right\} dx \tag{5.10.1}$$

The numerical formula for this double integration over a two-dimensional region takes the form

$$I(a, b, c(x), d(x)) = \sum_{m=1}^{M} w_m \sum_{n=1}^{N} v_n f(x_m, y_{m,n}) \tag{5.10.2}$$

where the weights w_m, v_n depend on the method of one-dimensional integration we choose.

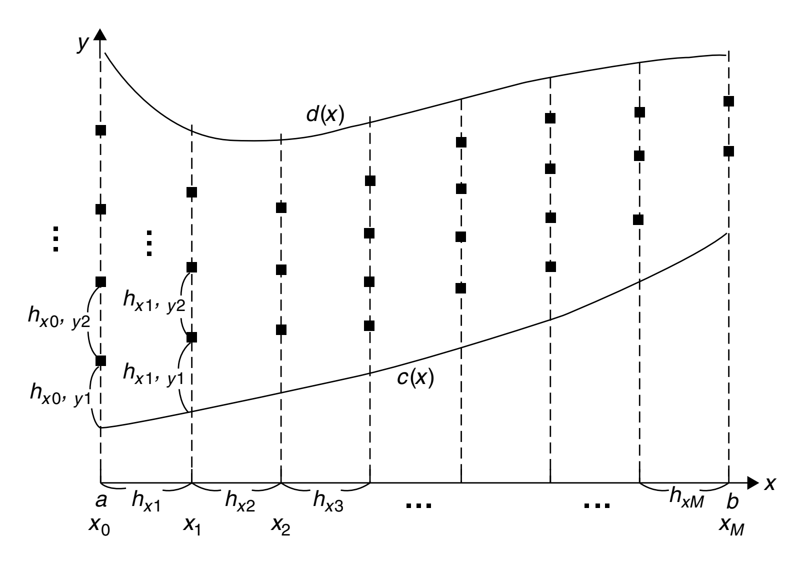


Figure 5.5 A region for a double integral.

(cf) The MATLAB built-in routine `dblquad()` can accept the boundaries of integration region only given as numbers. Therefore, if we want to use the routine in computing a double integral for a nonrectangular region D, we should define the integrand function $f(x, y)$ for a rectangular region $R \supseteq D$ (containing the actual integration region D) in such a way that $f(x, y) = 0$ for $(x, y) \notin D$; that is, the value of the function becomes zero outside the integration region D, which may result in more computations.

```
function INTfxy = int2s(f,a,b,c,d,M,N)
%double integral of f(x,y) over R = {(x,y)|a <= x <= b, c(x) <= y <= d(x)}
%  using Simpson's rule
if ceil(M) ~= floor(M) %fixed width of segments on x
   hx = M; M = ceil((b - a)/hx);
end
if mod(M,2) ~= 0, M = M + 1; end
hx = (b - a)/M; m = 1:M+1; x = a + (m - 1)*hx;
if isnumeric(c), cx(m) = c;   %if c is given as a constant number
 else  cx(m) = feval(c,x(m)); %in case c is given as a function of x
end
if isnumeric(d), dx(m) = d;   %if c is given as a constant number
 else  dx(m) = feval(d,x(m)); %in case d is given as a function of x
end
if ceil(N) ~= floor(N) %fixed width of segments on y
  hy = N; Nx(m) = ceil((dx(m)- cx(m))/hy);
  ind = find(mod(Nx(m),2) ~= 0); Nx(ind) = Nx(ind) + 1;
 else   %fixed number of subintervals
  if mod(N,2) ~= 0,  N = N +1;  end
  Nx(m) = N;
 end
for m = 1:M + 1
  sx(m) = smpsns_fxy(f,x(m),cx(m),dx(m),Nx(m));
end
kodd = 2:2:M; keven = 3:2:M - 1; %the set of odd/even indices
INTfxy = hx/3*(sx(1) + sx(M + 1) + 4*sum(sx(kodd)) + 2*sum(sx(keven)));
```

```
function INTf = smpsns_fxy(f, x, c, d, N)
%1-dimensional integration of f(x,y) for Ry = {c <= y <= d}
if nargin < 5, N = 100; end
if abs(d - c)< eps | N <= 0, INTf = 0; return; end
if mod(N,2) ~= 0,  N = N + 1; end
h = (d - c)/N; y = c+[0:N]*h; fxy = feval(f,x,y);
fxy(find(fxy == inf)) = realmax; fxy(find(fxy == -inf)) = -realmax;
kodd = 2:2:N; keven = 3:2:N - 1; %the set of odd/even indices
INTf = h/3*(fxy(1) + fxy(N + 1) + 4*sum(fxy(kodd)) + 2*sum(fxy(keven)));
```

```
%nm510: the volume of a sphere
x = [-1:0.05:1]; y = [0:0.05:1]; [X,Y] = meshgrid(x,y);
f510 = inline('sqrt(max(1 - x.*x - y.*y,0))','x','y');
Z = f510(X,Y); mesh(x,y,Z);
a = -1; b = 1; c = 0; d=inline('sqrt(max(1 - x.*x,0))','x');
Vs1 = int2s(f510,a,b,c,d,100,100) %with fixed number of segments
error1 = Vs1 - pi/3
Vs2 = int2s(f510,a,b,c,d,0.01,0.01) %with fixed segment width
error2 = Vs2 - pi/3
```

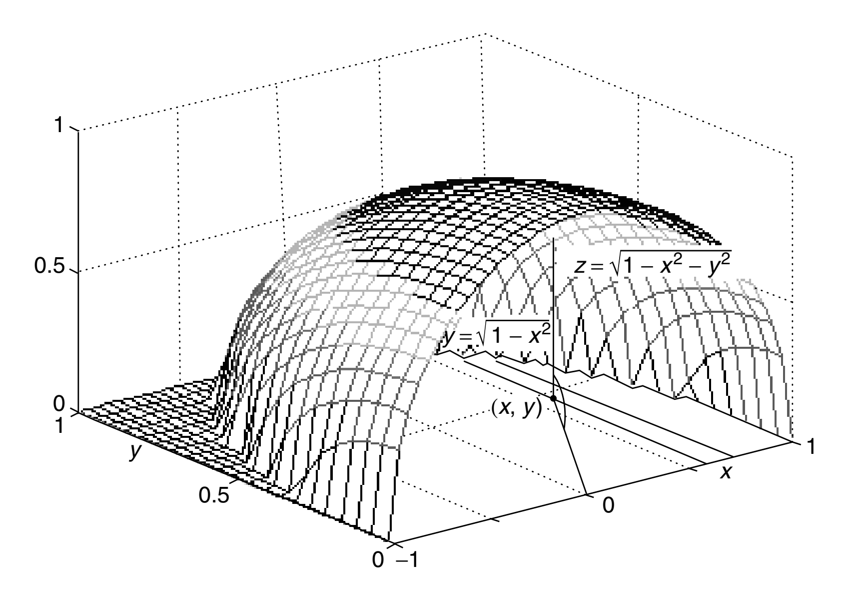


Figure 5.6 One-fourth (1/4) of a sphere with the radius $r = 1$.

Although the integration rules along the x axis and along the y axis do not need to be the same, we make a double integration routine "`int2s(f,a,b,c,d,M,N)`" which uses the Simpson method in common for both integrations and calls another routine "`smpsns_fxy()`" for one-dimensional integration along the y axis. The left/right boundary `a/b` of integration region given as the second/third input argument must be a number, while the lower/upper boundary `c/d` of integration region given as the fourth/fifth input argument may be either a number or a function of x. If the sixth/seventh input argument `M/N` is given as a positive integer, it will be accepted as the number of segments; otherwise, it will be interpreted as the segment width h_x/h_y. We also constructed a MATLAB program "`nm510`" in order to use the routine "`int2s()`" for finding one-fourth of the volume of a sphere with the radius $r = 1$ depicted in Fig. 5.6.

$$I = \int_{-1}^{1} \int_{0}^{\sqrt{1-x^2}} \sqrt{1 - x^2 - y^2}\, dy\, dx = \frac{\pi}{3} = 1.04719755\ldots \tag{5.10.3}$$

Interested readers are recommended to work with these routines and run the program "`nm510.m`" to see the result.

```
>>nm510
   Vs1 = 1.0470, error1 = -1.5315e-004
   Vs2 = 1.0470, error2 = -1.9685e-004
```

PROBLEMS

5.1 Numerical Differentiation of Basic Functions

If we want to find the derivative of a polynomial/trigonometric/exponential function, it would be more convenient and accurate to use an analytical computation (by hand) than to use a numerical computation (by computer). But, in order to test the accuracy of the numerical derivative formulas, consider the three basic functions as

$$f_1(x) = x^3 - 2x, \qquad f_2(x) = \sin x, \qquad f_3(x) = e^x \tag{P5.1.1}$$

(a) To find the first derivatives of these functions by using the formulas (5.1.8) and (5.1.9) listed in Table 5.3 (Section 5.3), modify the program "nm5p01.m", which uses the MATLAB routine "difapx()" (Section 5.3) for generating the coefficients of the numerical derivative formulas. Fill in the following table with the error results obtained from running the program.

First Derivatives	h	$\frac{f_1 - f_{-1}}{2h}$	$\frac{-f_2 + 8f_1 - 8f_{-1} + f_{-2}}{12h}$
$(x^3 - 2x)'\|_{x=1}$ $= 1.00000000$	0.1	1.0000e-02	
	0.01		9.1038e-15
$(\sin x)'\|_{x=\pi/3}$ $= 0.50000000$	0.1	8.3292e-04	
	0.01	8.3333e-06	
$(e^x)'\|_{x=0}$ $= 1.00000000$	0.1		3.3373e-06
	0.01	1.6667e-05	

```
%nm5p01
f = inline('x.*(x.*x-2)', 'x');
n = [1 -1];  x0 = 1;  h = 0.1;  DT = 1;
c = difapx(1,n);  i = 1:length(c);
num = c*feval(f,x0 + (n(1) + 1 - i)*h)'; drv = num/h;
fprintf('with h = %6.4f, %12.6f %12.4e\n', h,drv,drv - DT);
```

(b) Likewise in (a), modify the program "nm5p01.m" in such a way that the formulas (5.3.1) and (5.3.2) in Table 5.3 are generated and used to find the second numerical derivatives. Fill in the following table with the error results obtained from running the program.

Second Derivatives	h	$\frac{f_1 - 2f_0 + f_{-1}}{h^2}$	$\frac{-f_2 + 16f_1 - 30f_0 + 16f_{-1} - f_{-2}}{12h^2}$
$(x^3 - 2x)^{(2)}\|_{x=1}$ $= 6.0000000000$	0.1	2.6654e-14	
	0.01		2.9470e-12
$(\sin x)^{(2)}\|_{x=\pi/3}$ $= -0.8660254037$	0.1		9.6139e-07
	0.01	7.2169e-06	
$(e^x)^{(2)}\|_{x=0}$ $= 1.0000000000$	0.1	8.3361e-04	
	0.01		1.1183e-10

5.2 Numerical Differentiation of a Function Given as a Set of Data Pairs

Consider the three (numerical) functions each given as a set of five data pairs in Table P5.2.

Table P5.2 Three Functions Each Given as a Set of Five Data Pairs

x	$f_1(x)$	x	$f_2(x)$	x	$f_3(x)$
0.8000	−1.0880	0.8472	0.7494	−0.2000	1.2214
0.9000	−1.0710	0.9472	0.8118	−0.1000	1.1052
1.0000	−1.0000	1.0472	0.8660	0	1.0000
1.1000	−0.8690	1.1472	0.9116	0.1000	0.9048
1.2000	−0.6720	1.2472	0.9481	0.2000	0.8187

(a) Use the formulas (5.1.8) and (5.1.9) to find the first derivatives of the three numerical functions (at $x = 1$, 1.0472 and 0, respectively) and fill in the following table with the results. Also use the formulas (5.3.1) and (5.3.2) to find the second derivatives of the three functions (at $x = 1$, 1.0472 and 0, respectively) and fill in the following table with the results.

	$f_1'(x)\|_{x=1}$	$f_2'(x)\|_{x=1.0472}$	$f_3'(x)\|_{x=0}$
First derivative by Eq. (5.1.8)	1.0000e-02		2.0000e-03
First derivative by Eq. (5.1.9)		2.5000e-04	
	$f_1^{(2)}(x)\|_{x=1}$	$f_2^{(2)}(x)\|_{x=1.0472}$	$f_3^{(2)}(x)\|_{x=0}$
Second derivative by Eq. (5.3.1)		6.0254e-03	
Second derivative by Eq. (5.3.2)	2.4869e-14		8.3333e-04

(b) Based on the Lagrange/Newton polynomial matching the three/five points around the target point, find the first/second derivatives of the three functions (at $x = 1$, 1.0472 and 0, respectively) and fill in the following table with the results.

	$f_1'(x)\|_{x=1}$	$f_2'(x)\|_{x=1.0472}$	$f_3'(x)\|_{x=0}$
First derivative on $l_2(x)$		1.0000e-03	
First derivative on $l_4(x)$	4.3201e-12		4.1667e-04
	$f_1^{(2)}(x)\|_{x=1}$	$f_2^{(2)}(x)\|_{x=1.0472}$	$f_3^{(2)}(x)\|_{x=0}$
Second derivative on $l_2(x)$	1.4211e-14		0.0000e+00
Second derivative on $l_4(x)$		6.8587e-03	

5.3 First Derivative and Step-size

Consider the routine "`jacob()`" in Section 4.6, which is used for computing the Jacobian—that is, the first derivative of a vector function with respect to a vector variable.

(a) Which one is used for computing the Jacobian in the routine "`jacob()`" among the first derivative formulas in Section 5.1?

(b) Expecting that smaller step-size h would yield a better solution to the problem given in Example 4.3, Bush changed `h = 1e-4` to `h = 1e-5` in the routine "`newtons()`" and then typed the following statement into the MATLAB command window. What solution could he get?

```
>>rn1 = newtons('phys',1e6,1e-4,100)
```

(c) What baffled him out of his expectation? Jessica diagnosed the trouble as caused by a singular Jacobian matrix and modified the statement '`dx = -jacob()\fx(:)`' in the routine "`newtons()`" as follows. What solution (to the problem in Example 4.3) do you get by using the modified routine, that is, by typing the same statement as in (b)?

```
>>rn2 = newtons('phys',1e6,1e-4,100), phys(rn2)
```

```
J = jacob(f,xx(k,:),h,varargin{:});
if rank(J) < Nx
  k = k - 1;
  fprintf('Jacobian singular! det(J) = %12.6e\n',det(J)); break;
 else
  dx = -J\fx(:); %-[dfdx]^-1*fx;
end
```

(d) To investigate how the accident of Jacobian singularity happened, add `h = 1e-5` to the (tentative) solution (`rn2`) obtained in (c). Does the result differ from `rn2`? If not, why? (See Section 1.2.2 and Problem 1.19.)

```
>>rn2 + 1e-5 ~= rn2
```

(e) Charley thought that Jessica just circumvented the Jacobian singularity problem. To remove the source of singularity, he modified the formula (5.1.8) into

$$D_{c2}^{'}(x,h) = \frac{f((1+h)x) - f((1-h)x)}{2hx} \tag{P5.5.3}$$

and implemented it in another routine "`jacob1()`" as follows.

```
function g = jacob1(f,x,h,varargin) %Jacobian of f(x)
if nargin<3, h =.0001; end
h2 = 2*h; N = length(x); I = eye(N);
for n = 1:N
 if abs(x(n))<.0001, x(n) =.0001; end
   delta = h*x(n);
   tmp = I(n,:)*delta;
   f1 = feval(f,x + tmp,varargin{:});
   f2 = feval(f,x - tmp,varargin{:});
   f12 = (f1 - f2)/2/delta; g(:,n) = f12(:);
end
```

With `h = 1e-5` or `h = 1e-6` and `jacob()` replaced by `jacob1()` in the routine "`newtons()`", type the same statement as in (c) to get a solution to the problem in Example 4.3 together with its residual error and check if his scheme works fine.

```
>>rn3 = newtons('phys',1e6,1e-4,100), phys(rn3)
```

5.4 Numerical Integration of Basic Functions

Compute the following integrals by using the trapezoidal rule, the Simpson's rule, and Romberg method and fill in the following table with the resulting errors.

(i) $\int_0^2 (x^3 - 2x)\,dx$ **(ii)** $\int_0^{\pi/2} \sin x\,dx$ **(iii)** $\int_0^1 e^{-x}\,dx$

	N	Trapezoidal Rule	Simpson Rule	Romberg (tol = 0.0005)
$\int_0^2 (x^3 - 2x)\,dx = 0$	4		0.0000e+0	
	8	6.2500e-1		
$\int_0^{\pi/2} \sin x\,dx = 1$	4	1.2884e-2		8.4345e-6
	8		8.2955e-6	
$\int_0^1 e^{-x}\,dx = 0.63212055883$	4		1.3616e-5	
	8	8.2286e-4		

5.5 Adaptive Quadrature and Gaussian Quadrature for Improper Integral

Consider the following two integrals.

(i) $$\int_0^1 \frac{1}{\sqrt{x}}\,dx = 2x^{1/2}\Big|_0^1 = 2 \tag{P5.5.1}$$

(ii) $$\int_{-1}^1 \frac{1}{\sqrt{x}}\,dx = \int_{-1}^0 \frac{1}{\sqrt{x}}\,dx + \int_0^1 \frac{1}{\sqrt{x}}\,dx = 2 - 2i \tag{P5.5.2}$$

(a) Type the following statements into the MATLAB command window to use the integration routines for the above integral. What did you get? If something is wrong, what do you think caused it?

```
>>f = inline('1./sqrt(x)','x'); % define the integrand function
>>smpsns(f,0,1,100) % integral over [0,1] with 100 segments
>>rmbrg(f,0,1,1e-4) % with error tolerance = 0.0001
>>adapt_smpsn(f,0,1,1e-4) % with error tolerance = 0.0001
>>gauss_legendre(f,0,1,20) %Gauss-Legendre with N = 20 grid points
>>quad(f,0,1) % MATLAB built-in routine
>>quad8(f,0,1) % MATLAB 5.x built-in routine
>>adapt_smpsn(f,-1,1,1e-4) %integral over [-1,1]
>>quad(f,-1,1) % MATLAB built-in routine
>>quadl(f,-1,1) % MATLAB built-in routine
```

(b) Itha decided to retry the routine "`smpsns()`", but with the singular point excluded from the integration interval. In order to do that, she replaced the singular point (0) which is the lower bound of the integration interval [0,1] by 10^{-4} or 10^{-5}, and typed the following statements into the MATLAB command window.

```
>>smpsns(f,1e-4,1,100)
>>smpsns(f,1e-5,1,100)
>>smpsns(f,1e-5,1,1e4)
>>smpsns(f,1e-4,1,1e3)
>>smpsns(f,1e-4,1,1e4)
```

What are the results? Will it be better if you make the lower-bound of the integration interval closer to zero (0), without increasing the number of segments or (equivalently) decreasing the segment width? How about increasing the number of segments without making the lower bound of the integration interval closer to the original lower-bound which is zero (0)?

(c) For the purpose of improving the performance of "`adap_smpsn()`", Vania would put the following statements into both of the routines "`smpsns()`" and "`adap_smpsn()`". Supplement the routines and check whether her idea works or not.

```
EPS = 1e-12; fa = feval(f,a,varargin{:});
if isnan(fa)|abs(fa) == inf, a = a + max(abs(a)*EPS,EPS); end
fb = feval(f,b,varargin{:});
?? ??????????????? ?? ????? ? ?? ? ? ???????????????????? ???
```

5.6 Various Numerical Integration Methods and Improper Integral

Consider the following integrals.

$$\int_0^{\infty} \frac{\sin x}{x}\,dx = \frac{\pi}{2} \cong \int_0^{100} \frac{\sin x}{x}\,dx \tag{P5.6.1}$$

$$\int_0^{\infty} e^{-x^2}\,dx = \frac{1}{2}\sqrt{\pi} \tag{P5.6.2}$$

Note that the true values of these integrals can be obtained by using the symbolic computation command "`int()`" as below.

```
>>syms x, int(sin(x)/x,0,inf)
>>int(exp(-x^2),0,inf)
```

(cf) Don't you believe it without seeing it? Blessed are those who have not seen and yet believe.

(a) To apply the routines like "`smpsns()`", "`adapt_smpsn()`", "`Gauss_Legendre()`" and "`quadl()`" for evaluating the integral (P5.6.1), do the following.

(i) Note that the integration interval $[0, \infty)$ can be changed into a finite interval as below.

$$\begin{aligned}\int_0^{\infty} \frac{\sin x}{x}\,dx &= \int_0^{1} \frac{\sin x}{x}\,dx + \int_1^{\infty} \frac{\sin x}{x}\,dx \\ &= \int_0^{1} \frac{\sin x}{x}\,dx + \int_1^{0} \frac{\sin(1/y)}{1/y}\left(-\frac{1}{y^2}\right)dy \\ &= \int_0^{1} \frac{\sin x}{x}\,dx + \int_0^{1} \frac{\sin(1/y)}{y}\,dy\end{aligned} \tag{P5.6.3}$$

(ii) Add the block of statements in P5.5(c) into the routines "`smpsns()`" and "`adap_smpsn()`" to make them cope with the cases of `NaN` (Not-a-Number) and `Inf` (Infinity).

(iii) Supplement the program "`nm5p06a.m`" so that the various routines are applied for computing the integrals (P5.6.1) and (P5.6.3), where the parameters like the number of segments ($N = 200$), the error tolerance (tol = 1e-4), and the number of grid points (MGL = 20) are supposed to be used as they are in the program. Noting that the second integrand function in (P5.6.3) oscillates like crazy with higher frequency and larger amplitude as y gets closer to zero (0), set the lower bound of the integration interval to `a2 = 0.001`.

(iv) Run the supplemented program and fill in Table P5.6 with the absolute errors of the results.

```
%nm5p06a
warning off MATLAB:divideByZero
fp56a = inline('sin(x)./x','x'); fp56a2 = inline('sin(1./y)./y','y');
IT = pi/2; % True value of the integral
a = 0; b = 100; N = 200; tol = 1e-4; MGL = 20; a1 = 0; b1 = 1; a2 = 0.001; b2 = 1;
format short e
e_s = smpsns(fp56a,a,b,N)-IT
e_as = adapt_smpsn(fp56a,a,b,tol)-IT
e_ql = quadl(fp56a,a,b,tol)-IT
e_GL = Gauss_Legendre(fp56a,a,b,MGL)-IT
e_ss = smpsns(fp56a,a1,b1,N) + smpsns(fp56a2,a2,b2,N)-IT
e_Iasas = adapt_smpsn(fp56a,a1,b1,tol)+ ...
   ???????????????????????????? -IT
e_Iqq = quad(fp56a,a1,b1,tol)+??????????????????????????? -IT
warning on MATLAB:divideByZero
```

```
%nm5p06b
warning off MATLAB:divideByZero
fp56b = inline('exp(-x.*x)','x');
fp56b1 = inline('ones(size(x))','x');
fp56b2 = inline('exp(-1./y./y)./y./y','y');
a = 0; b = 200; N = 200; tol = 1e-4; IT = sqrt(pi)/2;
a1 = 0; b1 = 1; a2 = 0; b2 = 1; MGH = 2;
e_s = smpsns(fp56b,a,b,N)-IT
e_as = adapt_smpsn(fp56b,a,b,tol)-IT
e_q = quad(fp56b,a,b,tol)-IT
e_GH = Gauss_Hermite(fp56b1,MGH)/2-IT
e_ss = smpsns(fp56b,a1,b1,N) + smpsns(fp56b2,a2,b2,N)-IT
Iasas = adapt_smpsn(fp56b,a1,b1,tol)+ ...
       +????????????????????????????? -IT
e_qq = quad(fp56b,a1,b1,tol)+????????????????????????? -IT
warning off MATLAB:divideByZero
```

Table P5.6 Results of Applying Various Numerical Integration Methods for Improper Integrals

	Simpson	adaptive	quad	Gauss	S&S	a&a	q&q
(P5.6.1)	8.5740e-3		1.9135e-1		1.1969e+0		2.4830e-1
(P5.6.2)		6.6730e-6		0.0000e+0		3.3546e-5	

(b) To apply the routines like "`smpsns()`", "`adapt_smpsn()`", "`quad()`", and "`Gauss_Hermite()`" for evaluating the integral (P5.6.2), do the following.

(i) Note that the integration interval $[0, \infty)$ can be changed into a finite interval as below.

$$\int_0^{\infty} e^{-x^2}\,dx = \int_0^1 e^{-x^2}\,dx + \int_1^{\infty} e^{-x^2}\,dx$$
$$= \int_0^1 e^{-x^2}\,dx + \int_1^0 e^{-1/y^2}\left(-\frac{1}{y^2}\right)dy$$
$$= \int_0^1 e^{-x^2}\,dx + \int_0^1 \frac{e^{-1/y^2}}{y^2}\,dy \qquad \text{(P5.6.4)}$$

(ii) Compose the incomplete routine "`Gauss_Hermite`" like "`Gauss_Legendre`", which performs the Gauss–Hermite integration introduced in Section 5.9.2.

(iii) Supplement the program "`nm5p06b.m`" so that the various routines are applied for computing the integrals (P5.6.2) and (P5.6.4), where the parameters like the number of segments ($N = 200$), the error tolerance (tol = 1e-4) and the number of grid points (MGH = 2) are supposed to be used as they are in the program. Note that the integration interval is not $(-\infty, \infty)$ like that of Eq. (5.9.12), but $[0, \infty)$ and so you should cut the result of "`Gauss_Hermite()`" by half to get the right answer for the integral (P5.6.2).

(iv) Run the supplemented program and fill in Table P5.6 with the absolute errors of the results.

(c) Based on the results listed in Table P5.6, answer the following questions:

(i) Among the routines "`smpsns()`", "`adapt_smpsn()`", "`quad()`", and "`Gauss()`", choose the best two ones for (P5.6.1) and (P5.6.2), respectively.

(ii) The routine "`Gauss–Legendre()`" works (badly, perfectly) even with as many as 20 grid points for (P5.6.1), while the routine

"`Gauss_Hermite()`" works (perfectly, badly) just with two grid points for (P5.6.2). It is because the integrand function of (P5.6.1) is (far from, just like) a polynomial, while (P5.6.2) matches Eq. (5.9.11) and the part of it excluding e^{-x^2} is (just like, far from) a polynomial.

```
function I = Gauss_Hermite(f,N,varargin)
[t,w]=???????(N);
ft = feval(f,t,varargin{:});
I = w*ft';
```

(iii) Run the following program "`nm5p06c.m`" to see the shapes of the integrand functions of (P5.6.1) and (P5.6.2) and the second integral of (P5.6.3). You can zoom in/out the graphs by clicking the Tools/Zoom_in menu and then clicking any point on the graphs with the left/right mouse button in the MATLAB graphic window. Which one is oscillating furiously? Which one is oscillating moderately? Which one is just changing abruptly?

```
%nm5p06c
clf
fp56a = inline('sin(x)./x','x');
fp56a2 = inline('sin(1./y)./y','y');
fp56b = inline('exp(-x.*x)','x');
x0 = [eps:2000]/20; x = [eps:100]/100;
subplot(221), plot(x0,fp56a(x0))
subplot(223), plot(x0,fp56b(x0))
subplot(222), y = logspace(-3,0,2000); loglog(y,abs(fp56a2(y)))
subplot(224), y = logspace(-6,-3,2000); loglog(y,abs(fp56a2(y)))
```

(iv) The adaptive integration routines like "`adapt_smpsn()`" and "`quad()`" work (badly, fine) for (P5.6.1), but (fine, badly) for (P5.6.2). From this fact, we might conjecture that the adaptive integration routines may be (ineffective, effective) for the integrand functions which have many oscillations, while they may be (effective, ineffective) for the integrand functions which have abruptly changing slope. To support this conjecture, run the following program "`nm5p06d`", which uses the "`quad()`" routine for the integrals

$$\int_1^b \frac{\sin x}{x}\,dx \qquad \text{with } b = 100, 1000, 10000\ldots. \qquad \text{(P5.6.5a)}$$

$$\int_a^1 \frac{\sin(1/y)}{y}\,dy \qquad \text{with } a = 0.001, 0.0001, 0.00001, \ldots \text{(P5.6.5b)}$$

```
%nm5p06d
fp56a = inline('sin(x)./x','x');
fp56a2 = inline('sin(1./y)./y','y');
syms x
IT2 = pi/2 - double(int(sin(x)/x,0,1)) %true value of the integral
disp('Change of upper limit of the integration interval')
a = 1; b = [100 1e3 1e4 1e7]; tol = 1e-4;
for i = 1:length(b)
  Iq2 = quad(fp56a,a,b(i),tol);
  fprintf('With b = %12.4e, err_Iq = %12.4e\n', b(i),Iq2-IT2);
end
disp('Change of lower limit of the integration interval')
a2 = [1e-3 1e-4 1e-5 1e-6 0]; b2 = 1; tol = 1e-4;
for i = 1:5
  Iq2 = quad(fp56a2,a2(i),b2,tol);
  fprintf('With a2=%12.4e, err_Iq=%12.4e\n', a2(i),Iq2-IT2);
end
```

Does the "`quad()`" routine work stably for (P5.6.5a) with the changing value of the upper-bound of the integration interval? Does it work stably for (P5.6.5b) with the changing value of the lower-bound of the integration interval? Do the results support or defy the conjecture?

(cf) This problem warns us that it may be not good to use only one routine for a computational work and suggests us to use more than one method for cross check.

5.7 Gauss–Hermite Integration Method

Consider the following integral:

$$\int_0^\infty e^{-x^2} \cos x \, dx = \frac{\sqrt{\pi}}{2} e^{-1/4} \tag{P5.7.1}$$

Select a Gauss quadrature suitable for this integral and apply it with the number of grid points `N = 4` as well as the routines "`smpsns()`", "`adapt_smpsn()`", "`quad()`", and "`quadl()`" to evaluate the integral. In order to compare the number of floating-point operations required to achieve almost the same level of accuracy, set the number of segments for Simpson method to `N` = 700 and the error tolerance for all other routines to `tol` = 10^{-5}. Fill in Table P5.7 with the error results.

Table P5.7 The Results of Applying Various Numerical Integration Methods

		Simpson (`N` = 700)	adaptive (`tol` = 10^{-5})	Gauss	quad (`tol` = 10^{-5})	quadl (`tol` = 10^{-5})
(P5.7.1)	|error|		1.0001e-3		1.0000e-3	
	flops	4930	5457	1484	11837	52590 (with quad8)
(P5.8.1)	|error|	1.3771e-2		0		4.9967e-7
	flops	5024	7757	131	28369	75822

5.8 Gauss–Laguerre Integration Method

(a) As in Section 5.9.1, Section 5.9.2, and Problem 5.6(b), compose the MATLAB routines: "`Laguerp()`", which generates the Laguerre polynomial (5.9.18); "`Gausslgp()`", which finds the grid point t_i's and the coefficient $w_{N,i}$'s for Gauss–Laguerre integration formula (5.9.16); and "`Gauss_Laguerre(f,N)`", which uses these two routines to carry out the Gauss–Laguerre integration method.

(b) Consider the following integral:

$$\int_0^\infty e^{-t} t\, dt = -e^{-t} t \Big|_0^\infty + \int_0^\infty e^{-t}\, dt = -e^{-t}\Big|_0^\infty = 1 \qquad \text{(P5.8.1)}$$

Noting that, since this integral matches Eq. (5.9.17) with $f(t) = t$, Gauss–Laguerre method is the right choice, apply the routine "`Gauss_Laguerre(f,N)`" (manufactured in (a)) with `N` $= 2$ as well as the routines "`smpsns()`", "`adapt_smpsn()`", "`quad()`", and "`quadl()`" for evaluating the integral and fill in Table P5.7 with the error results. Which turns out to be the best? Is the performance of "`quad()`" improved by lowering the error tolerance?

(cf) This illustrates that the routine "`adapt_smpsn()`" sometimes outperforms the MATLAB built-in routine "`quad()`" with fewer computations. On the other hand, Table P5.7 shows that it is most desirable to apply the Gauss quadrature schemes only if one of them is applicable to the integration problem.

5.9 Numerical Integrals

Consider the following integrals.

(1) $\int_0^{\pi/2} x \sin x\, dx = 1$

(2) $\int_0^1 x \ln(\sin x)\, dx = -\frac{1}{2}\pi^2 \ln 2$

(3) $\int_0^1 \frac{1}{x(1-\ln x)^2}\, dx = 1$

(4) $\int_1^\infty \frac{1}{x(1+\ln x)^2}\, dx = 1$

(5) $\int_0^1 \frac{1}{\sqrt{x}(1+x)}\, dx = \frac{\pi}{2}$

(6) $\int_1^\infty \frac{1}{\sqrt{x}(1+x)}\, dx = \frac{\pi}{2}$

(7) $\int_0^1 \sqrt{\ln \frac{1}{x}}\, dx = \frac{\sqrt{\pi}}{2}$

(8) $\int_0^\infty \sqrt{x} e^{-x}\, dx = \frac{\sqrt{\pi}}{2}$

(9) $\int_0^\infty x^2 e^{-x} \cos x\, dx = -\frac{1}{2}$

(a) Apply the integration routines "`smpsns()`" (with $N = 10^4$), "`adapt_smpsn()`", "`quad()`", "`quadl()`" (tol $= 10^{-6}$) and "`Gauss_legendre()`" (Section 5.9.1) or "`Gauss_Laguerre()`" (Problem 5.8) (with `N` $= 15$) to compute the above integrals and fill in Table P5.9 with the relative errors. Use the upper/lower bounds of the integration interval in Table P5.9 if they are specified in the table.

(b) Based on the results listed in Table P5.9, answer the following questions or circle the right answer.

(i) From the fact that the Gauss–Legendre integration scheme worked best only for (1), it is implied that the scheme is (recommendable, not recommendable) for the case where the integrand function is far from being approximated by a polynomial.

(ii) From the fact that the Gauss–Laguerre integration scheme worked best only for (9), it is implied that the scheme is (recommendable, not recommendable) for the case where the integrand function excluding the multiplying term e^{-x} is far from being approximated by a polynomial.

(iii) Note the following:

- The integrals (3) and (4) can be converted into each other by a variable substitution of $x = u^{-1}$, $dx = -u^{-2}\,du$. The integrals (5) and (6) have the same relationship.
- The integrals (7) and (8) can be converted into each other by a variable substitution of $u = e^{-x}$, $dx = -u^{-1}du$.

From the results for (3)–(8), it can be conjectured that the numerical integration may work (better, worse) if the integration interval is changed from $[1, \infty)$ into (0,1] through the substitution of variable like

$$x = u^{-n},\ dx = -nu^{-(n+1)}du \text{ or } u = e^{-nx},\ dx = -(nu)^{-1}\,du \tag{P5.9.1}$$

Table P5.9 The Relative Error Results of Applying Various Numerical Integration Methods

	Simpson ($N = 10^4$)	Adaptive ($tol = 10^{-6}$)	Gauss ($N = 10$)	quad ($tol = 10^{-6}$)	quadl ($tol = 10^{-6}$)
(1)	1.9984e-15		0.0000e+00		7.5719e-11
(2)		2.8955e-08		1.5343e-06	
(3)	9.7850e-02 ($a = 10^{-4}$)		1.2713e-01		2.2352e-02
(4), $b = 10^4$		9.7940e-02		9.7939e-02	
(5)	1.2702e-02 ($a = 10^{-4}$)		3.5782e-02		2.6443e-07
(6), $b = 10^3$		4.0250e-02		4.0250e-02	
(7)	6.8678e-05		5.1077e-04		3.1781e-07
(8), $b = 10$		1.6951e-04		1.7392e-04	
(9), $b = 10$	7.8276e-04		2.9237e-07		7.8276e-04

5.10 The BER (Bit Error Rate) Curve of Communication with Multidimensional Signaling

For a communication system with multidimensional (orthogonal) signaling, the BER—that is, the probability of bit error—is derived as

$$P_{e,b} = \frac{2^{b-1}}{2^b - 1}\left(1 - \frac{1}{\sqrt{\pi}}\int_{-\infty}^{\infty} (Q^{M-1}(-\sqrt{2}y - \sqrt{b\mathrm{SNR}}))e^{-y^2}\,dy\right) \tag{P5.10.1}$$

where b is the number of bits, $M = 2^b$ is the number of orthogonal waveforms, SNR is the signal-to-noise-ratio, and $Q(\cdot)$ is the error function defined by

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} e^{-y^2/2}\, dy \tag{P5.10.2}$$

We want to plot the BER curves for SNR = 0:10[dB] and b = 1:4.

(a) Consider the following program "`nm5p10.m`", whose objective is to compute the values of $P_{e,b}(\text{SNR},b)$ for SNR = 0:10[dB] and b = 1:4 by using the routine "`Gauss_Hermite()`" (Problem 5.6) and also by using the MATLAB built-in routine "`quad()`" and to plot them versus $\text{SNR[dB]} = 10\log_{10}\text{SNR}$. Complete the incomplete part which computes the integral in (P5.10.1) over $[-1000, 1000]$ and run the program to obtain the BER curves like Fig. P5.10.

(b) Of the two routines, which one is faster and which one presents us with more reliable values of the integral in (P5.10.1)?

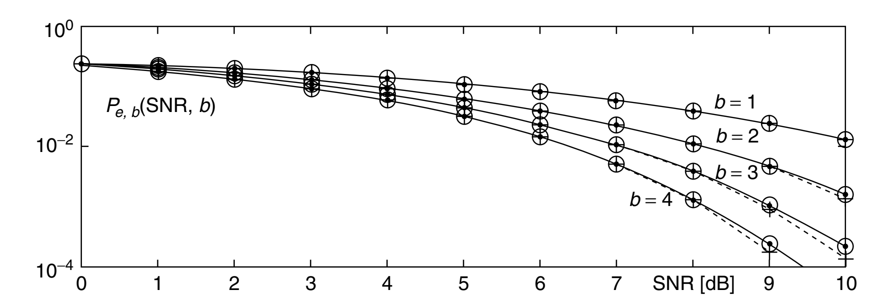


Figure P5.10 The BER (bit error rate) curves for multidimensional (orthogonal) signaling.

```
%nm5p10.m: plots the probability of bit error versus SNRbdB
fs ='Q(-sqrt(2)*x - sqrt(b*SNR)).^(2^b - 1)';
Q = inline('erfc(x/sqrt(2))/2','x');
f = inline(fs,'x','SNR','b');
fex2 = inline([fs '.*exp(-x.*x)'],'x','SNR','b');
SNRdB = 0:10; tol = 1e-4;  % SNR[dB] and tolerance used for 'quad'
for b = 1:4
  tmp = 2^(b - 1)/(2^b - 1); spi = sqrt(pi);
  for i = 1:length(SNRdB),
    SNR = 10^(SNRdB(i)/10);
    Pe(i) = tmp*(1-Gauss_Hermite(f,10,SNR,b)/spi);
    Pe1(i) = tmp*(1-quad(fex2,-10,10,tol,[],SNR,b)/spi);
    Pe2(i) = tmp*(1-?????????????????????????????????)/spi);
  end
  semilogy(SNRdB,Pe,'ko',SNRdB,Pe1,'b+:',SNRdB,Pe2,'r.-'), hold on
end
```

5.11 Length of Curve/Arc: Superb Harmony of Numerical Derivative/Integral.

The graph of a function $y = f(x)$ of a variable x is generally a curve and its length over the interval $[a, b]$ on the x-axis can be described by a line integral as

$$I = \int_a^b dl = \int_a^b \sqrt{dx^2 + dy^2} = \int_a^b \sqrt{1 + (dy/dx)^2}\, dx$$
$$= \int_a^b \sqrt{1 + (f'(x))^2}\, dx \qquad \text{(P5.11.1)}$$

For example, the length of the half-circumference of a circle with the radius of unit length can be obtained from this line integral with

$$y = f(x) = \sqrt{1 - x^2}, \qquad a = -1, \qquad b = 1 \qquad \text{(P5.11.2)}$$

Starting from the program "nm5p11.m", make a program that uses the numerical integration routines "smpsns()", "adapt_smpsn()", "quad()", "quadl()", and "Gauss_Legendre()" to evaluate the integral (P5.11.1,2) with the first derivative approximated by Eq. (5.1.8), where the parameters like the number of segments (N), the error tolerance (tol), and the number of grid points (M) are supposed to be as they are in the program. Run the program with the step size $h = 0.001$, 0.0001, and 0.00001 in the numerical derivative and fill in Table P5.11 with the errors of the results, noting that the true value of the half-circumference of a unit circle is π.

```
%nm5p11
a = -1; b = 1; % the lower/upper bounds of the integration interval
N = 1000 % the number of segments for the Simpson method
tol = 1e-6 % the error tolerance
M = 20 % the number of grid points for Gauss–Legendre integration
IT = pi; h = 1e-3 % true integral and step size for numerical derivative
flength = inline('sqrt(1 + dfp511(x,h).^2)','x','h');%integrand P5.11.1)
Is = smpsns(flength,a,b,N,h);
[Ias,points,err] = adapt_smpsn(flength,a,b,tol,h);
Iq = quad(flength,a,b,tol,[],h);
Iql = quadl(flength,a,b,tol,[],h);
IGL = Gauss_Legendre(flength,a,b,M,h);
```

```
function df = dfp511(x,h) % numerical derivative of (P5.11.2)
if nargin < 2,  h = 0.001;  end
df = (fp511(x + h)-fp511(x - h))/2/h; %Eq.(5.1.8)
```

```
function y = fp511(x)
y = sqrt(max(1-x.*x,0)); % the function (P5.11.2)
```

Table P5.11 Results of Applying Various Numerical Integration Methods for (P5.11.1,2)/(P5.12.1,2)

	Step-size h	Simpson	Adaptive	quad	quadl	Gauss
(P5.11.1,2)	0.001	4.6212e-2		2.9822e-2		8.4103e-2
	0.0001		9.4278e-3		9.4277e-3	
	0.00001	2.1853e-1		2.9858e-3		8.4937e-2
(P5.12.1,2)	0.001		1.2393e-5		1.3545e-5	
	0.0001	8.3626e-3		5.0315e-6		6.4849e-6
	0.00001		1.3846e-9		8.8255e-7	
(P5.13.1)	N/A	8.8818e-16		0		8.8818e-16

5.12 Surface Area of Revolutionary 3-D (Cubic) Object

The upper/lower surface area of a 3-D structure formed by one revolution of a graph (curve) of a function $y = f(x)$ around the x-axis over the interval $[a, b]$ can be described by the following integral:

$$I = 2\pi \int_a^b y\,dl = 2\pi \int_a^b f(x)\sqrt{1 + (f'(x))^2}\,dx \qquad \text{(P5.12.1)}$$

For example, the surface area of a sphere with the radius of unit length can be obtained from this equation with

$$y = f(x) = \sqrt{1 - x^2}, \qquad a = -1, \qquad b = 1 \qquad \text{(P5.12.2)}$$

Starting from the program "`nm5p11.m`", make a program "`nm5p12.m`" that uses the numerical integration routines "`smpsns()`" (with the number of segments `N = 1000`), "`adapt_smpsn()`", "`quad()`", "`quadl()`" (with the error tolerance `tol = 10`$^{-6}$) and "`Gauss_Legendre()`" (with the number of grid points `M = 20`) to evaluate the integral (P5.12.1,2) with the first derivative approximated by Eq. (5.1.8), where the parameters like the number of segments (`N`), the error tolerance (`tol`), and the number of grid points (`M`) are supposed to be as they are in the program. Run the program with the step size $h = 0.001$, 0.0001, and 0.00001 in the numerical derivative and fill in Table P5.11 with the errors of the results, noting that the true value of the surface area of a unit sphere is 4π .

5.13 Volume of Revolutionary 3-D (Cubic) Object

The volume of a 3-D structure formed by one revolution of a graph (curve) of a function $y = f(x)$ around the x-axis over the interval $[a, b]$ can be described by the following integral:

$$I = \pi \int_a^b f^2(x)\,dx \qquad \text{(P5.13.1)}$$

For example, the volume of a sphere with the radius of unit length (Fig. P5.13) can be obtained from this equation with Eq. (P5.12.2). Starting from the program "nm5p11.m", make a program "nm5p13.m" that uses the numerical integration routines "smpsns()" (with the number of segments N = 100), "adapt_smpsn()", "quad()", "quadl()" (with the error tolerance tol = 10^{-6}), and "Gauss_Legendre()" (with the number of grid points M = 2) to evaluate the integral (P5.13.1). Run the program and fill in Table P5.11 with the errors of the results, noting that the volume of a unit sphere is $4\pi/3$.

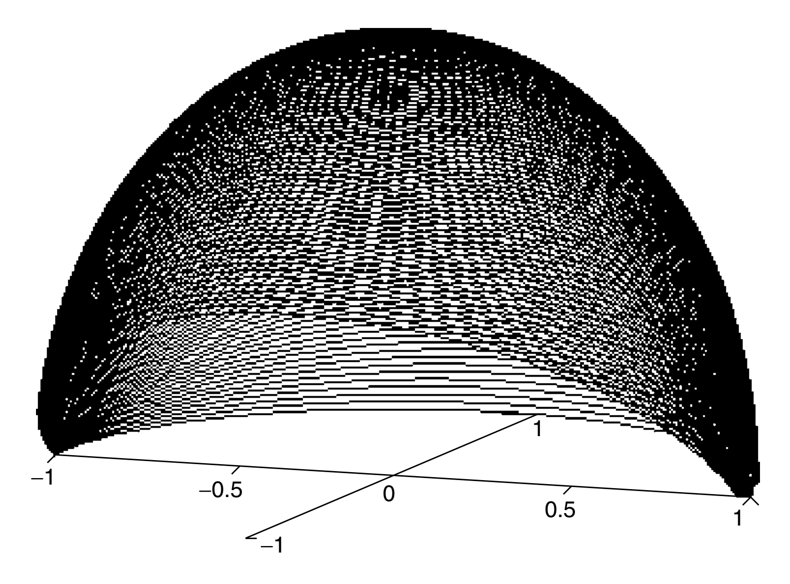


Figure P5.13 The surface and the volume of a unit sphere.

5.14 Double Integral

(a) Consider the following double integral

$$I = \int_0^2 \int_0^{\pi} y \sin x \, dx \, dy = \int_0^2 -y \cos x \Big|_0^{\pi} dy = \int_0^2 2y \, dy = y^2 \Big|_0^2 = 4 \tag{P5.14.1}$$

Use the routine "int2s()" (Section 5.10) with M = N = 20, M = N = 50 and M = N = 100 and the MATLAB built-in routine "dblquad()" to compute this double integral. Fill in Table P5.14.1 with the results and the times measured by using the commands tic/toc to be taken for carrying out each computation. Based on the results listed in Table P5.14.1, can we say that the numerical error becomes smaller as we increase the numbers (M,N) of segments along the x-axis and y-axis for the routine "int2s()"?

(b) Consider the following double integral:

$$I = \int_0^1 \int_0^1 \frac{1}{1 - xy}\, dx\, dy = \frac{\pi^2}{6} \qquad \text{(P5.14.2)}$$

Noting that the integrand function is singular at $(x, y) = (1, 1)$, use the routine "`int2s()`" and the MATLAB built-in routine "`dblquad()`" with the upper limit (`d`) of the integration interval along the y-axis `d` = 0.999, `d` = 0.9999, `d` = 0.99999 and `d` = 0.999999 to compute this double integral. Fill in Tables P5.14.2 and P5.14.3 with the results and the times measured by using the commands `tic`/`toc` to be taken for carrying out each computation.

Table P5.14.1 Results of Running "`int2s()`" and "`dblquad()`" for (P5.14.1)

	`int2s()`, `M = N = 20`	`int2s()`, `M = N = 100`	`int2s()`, `M = N = 200`	`dblquad()`
|error|		2.1649×10^{-8}		1.3250×10^{-8}
time				

Table P5.14.2 Results of Running "`int2s()`" and "`dblquad()`" for (P5.14.2)

<table>
<tr><td colspan="2"></td><td>a = 0, b = 1
c = 0,
d = 1-10⁻³</td><td>a = 0, b = 1
c = 0,
d = 1-10⁻⁴</td><td>a = 0, b = 1
c = 0,
d =1-10⁻⁵</td><td>a = 0, b = 1
c = 0,
d = 1-10⁻⁶</td></tr>
<tr><td rowspan="2">int2s()
M = 2000
N = 2000</td><td>|error|</td><td>0.0079</td><td></td><td>0.0024</td><td></td></tr>
<tr><td>time</td><td></td><td></td><td></td><td></td></tr>
<tr><td>dblquad</td><td>|error|</td><td></td><td>0.0004</td><td></td><td>0.0006</td></tr>
<tr><td></td><td>time</td><td></td><td></td><td></td><td></td></tr>
</table>

Table P5.14.3 Results of Running the Double Integral Routine "`int2s()`" for (P5.14.2)

<table>
<tr><td colspan="2"></td><td>M = 1000,
N = 1000</td><td>M = 2000,
N = 2000</td><td>M = 5000,
N = 5000</td></tr>
<tr><td rowspan="2">int2s()
a = 0, b = 1
c = 0, d = 1-10⁻⁴</td><td>|error|</td><td>0.0003</td><td></td><td></td></tr>
<tr><td>time</td><td></td><td></td><td></td></tr>
</table>

Based on the results listed in Tables P5.14.2 and P5.14.3, answer the following questions.

(i) Can we say that the numerical error becomes smaller as we set the upper limit (`d`) of the integration interval along the y-axis closer to the true limit 1?

(ii) Can we say that the numerical error becomes smaller as we increase the numbers (`M,N`) of segments along the x-axis and y-axis for the routine "`int2s()`"? If this is contrary to the case of (a), can you blame the weird shape of the integrand function in Eq. (P5.14.2) for such a mess-up?

(cf) Note that the computation times to be listed in Tables P5.14.1 to P5.14.3 may vary with the speed of CPU as well as the computational jobs which are concurrently processed by the CPU. Therefore, the time measured by the '`tic/toc`' commands cannot be an exact estimate of the computational load taken by each routine.

5.15 Area of a Triangle

Consider how to find the area between the graph (curve) of a function $f(x)$ and the x-axis. For example, let $f(x) = x$ for $0 \le x \le 1$ in order to find the area of a right-angled triangle with two equal sides of unit length. We might use either the 1-D integration or the 2-D integration—that is, the double integral for this job.

(a) Use any integration method that you like best to evaluate the integral

$$I_1 = \int_0^1 x\, dx = \frac{1}{2} \tag{P5.15.1}$$

(b) Use any double integration routine that you like best to evaluate the integral

$$I_2 = \int_0^1 \int_0^{f(x)} 1\, dy\, dx = \int_0^1 \int_0^x 1\, dy\, dx \tag{P5.15.2}$$

You may get puzzled with some problem when applying the routine "`int2s()`" if you define the integrand function as

```
>>fp515b = inline('1','x','y');
```

It is because this function, being called inside the routine "`smpsns_fxy()`", yields just a scalar output even for the vector-valued input argument. There are two remedies for this problem. One is to define the integrand function in such a way that it can generate the output of the same dimension as the input.

```
>>fp515b = inline('1+0*(x+y)','x','y');
```

But, this will cause a waste of computation time due to the dead multiplication for each element of the input arguments `x` and `y`. The other is to modify the routine "`smpsns_fxy()`" in such a way that it can avoid the vector operation. More specifically, you can replace some part of the routine with the following. But, this remedy also increases the computation time due to the abandonment of vector operation taking less time than scalar operation (see Section 1.3).

```
function INTf = smpsns_fxy(f,x,c,d,N)
.. .. .. .. .. .. .. .. .. .. ..
sum_odd = f(x,y(2));  sum_even = 0;
for n = 4:2:N
   sum_odd = sum_odd + f(x,y(n)); sum_even = sum_even + f(x,y(n - 1));
end
INTf = (f(x,y(1)) + f(x,y(N + 1)) + 4*sum_odd + 2*sum_even)*h/3;
.. .. .. .. .. .. .. .. .. .. ..
```

(cf) This problem illustrates that we must be provident to use the vector operation, especially in defining a MATLAB function.

5.16 Volume of a Cone

Likewise in Section 5.10, modify the program "`nm510.m`" so that it uses the routines "`int2s()`" and "`dblquad()`" to compute the volume of a cone that has a unit circle as its base side and a unit height, and run it to obtain the values of the volume up to four digits below the decimal point.)

6

ORDINARY DIFFERENTIAL EQUATIONS

Differential equations are mathematical descriptions of how the variables and their derivatives (rates of change) with respect to one or more independent variable affect each other in a dynamical way. Their solutions show us how the dependent variable(s) will change with the independent variable(s). Many problems in natural sciences and engineering fields are formulated into a scalar differential equation or a vector differential equation—that is, a system of differential equations.

In this chapter, we look into several methods of obtaining the numerical solutions to ordinary differential equations (ODEs) in which all dependent variables (x) depend on a single independent variable (t). First, the initial value problems (IVPs) will be handled with several methods including Runge–Kutta method and predictor–corrector methods in Sections 6.1 to 6.5. The final section (Section 6.6) will introduce the shooting method and the finite difference method for solving the two-point boundary value problem (BVP). ODEs are called an IVP if the values $x(t_0)$ of dependent variables are given at the initial point t_0 of the independent variable, while they are called a BVP if the values $x(t_0)/\ x(t_f)$ are given at the initial/final points t_0 and t_f.

6.1 EULER'S METHOD

When talking about the numerical solutions to ODEs, everyone starts with the Euler's method, since it is easy to understand and simple to program. Even though its low accuracy keeps it from being widely used for solving ODEs, it gives us a

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

clue to the basic concept of numerical solution for a differential equation simply and clearly. Let's consider a first-order differential equation:

$$y'(t) + a\ y(t) = r \qquad \text{with } y(0) = y_0 \tag{6.1.1}$$

It has the following form of analytical solution:

$$y(t) = \left(y_0 - \frac{r}{a}\right) e^{-at} + \frac{r}{a} \tag{6.1.2}$$

which can be obtained by using a conventional method or the Laplace transform technique [K-1, Chapter 5]. However, such a nice analytical solution does not exist for every differential equation; even if it exists, it is not easy to find even by using a computer equipped with the capability of symbolic computation. That is why we should study the numerical solutions to differential equations.

Then, how do we translate the differential equation into a form that can easily be handled by computer? First of all, we have to replace the derivative $y'(t) = dy/dt$ in the differential equation by a numerical derivative (introduced in Chapter 5), where the step-size h is determined based on the accuracy requirements and the computation time constraints. Euler's method approximates the derivative in Eq. (6.1.1) with Eq. (5.1.2) as

$$\frac{y(t+h) - y(t)}{h} + a\ y(t) = r$$

$$y(t+h) = (1 - ah)y(t) + hr \quad \text{with } y(0) = y_0 \tag{6.1.3}$$

and solves this difference equation step-by-step with increasing t by h each time from $t = 0$.

$$\begin{aligned} y(h) &= (1 - ah)y(0) + hr = (1 - ah)y_0 + hr \\ y(2h) &= (1 - ah)y(h) + hr = (1 - ah)^2 y_0 + (1 - ah)hr + hr \\ y(3h) &= (1 - ah)y(2h) + hr = (1 - ah)^3 y_0 + \textstyle\sum_{m=0}^{2} (1 - ah)^m hr \\ &\cdots\cdots\cdots\cdots\cdots\cdots \end{aligned} \tag{6.1.4}$$

This is a numeric sequence $\{y(kh)\}$, which we call a numerical solution of Eq. (6.1.1).

To be specific, let the parameters and the initial value of Eq. (6.1.1) be $a = 1$, $r = 1$, and $y_0 = 0$. Then, the analytical solution (6.1.2) becomes

$$y(t) = 1 - e^{-at} \tag{6.1.5}$$

```
%nm610: Euler method to solve a 1st-order differential equation
clear, clf
a = 1; r = 1; y0 = 0;  tf = 2;
t = [0:0.01:tf]; yt = 1 - exp(-a*t); %Eq.(6.1.5): true analytical solution
plot(t,yt,'k'), hold on
klasts = [8 4 2];  hs = tf./klasts;
y(1) = y0;
for itr = 1:3 %with various step size h = 1/8,1/4,1/2
   klast = klasts(itr); h = hs(itr); y(1)=y0;
   for k = 1:klast
      y(k + 1) = (1 - a*h)*y(k) +h*r; %Eq.(6.1.3):
      plot([k - 1 k]*h,[y(k) y(k+1)],'b', k*h,y(k+1),'ro')
      if k < 4, pause; end
   end
end
```

and the numerical solution (6.1.4) with the step-size $h = 0.5$ and $h = 0.25$ are as listed in Table 6.1 and depicted in Fig. 6.1. We make a MATLAB program "`nm610.m`", which uses Euler's method for the differential equation (6.1.1), actually solving the difference equation (6.1.3) and plots the graphs of the numerical solutions in Fig. 6.1. The graphs seem to tell us that a small step-size helps reduce the error so as to make the numerical solution closer to the (true) analytical solution. But, as will be investigated thoroughly in Section 6.2, it is only partially true. In fact, a too small step-size not only makes the computation time longer (proportional as $1/h$), but also results in rather larger errors due to the accumulated round-off effect. This is why we should look for other methods to decrease the errors rather than simply reduce the step-size.

Euler's method can also be applied for solving a first-order vector differential equation

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}) \qquad \text{with } \mathbf{y}(t_0) = \mathbf{y}_0 \tag{6.1.6}$$

which is equivalent to a high-order scalar differential equation. The algorithm can be described by

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(t_k, \mathbf{y}_k) \qquad \text{with } \mathbf{y}(t_0) = \mathbf{y}_0 \tag{6.1.7}$$

Table 6.1 A Numerical Solution of the Differential Equation (6.1.1) Obtained by the Euler's Method

t	$h = 0.5$	$h = 0.25$
0.25		$y(0.25) = (1 - ah)y_0 + hr = 1/4 = 0.25$
0.50	$y(0.50) = (1 - ah)y_0 + hr = 1/2 = 0.5$	$y(0.50) = (3/4)y(0.25) + 1/4 = 0.4375$
0.75		$y(0.75) = (3/4)y(0.50) + 1/4 = 0.5781$
1.00	$y(1.00) = (1/2)y(0.5) + 1/2 = 3/4 = 0.75$	$y(1.00) = (3/4)y(0.75) + 1/4 = 0.6836$
1.25		$y(1.25) = (3/4)y(1.00) + 1/4 = 0.7627$
1.50	$y(1.50) = (1/2)y(1.0) + 1/2 = 7/8 = 0.875$	$y(1.50) = (3/4)y(1.25) + 1/4 = 0.8220$
...	....	....

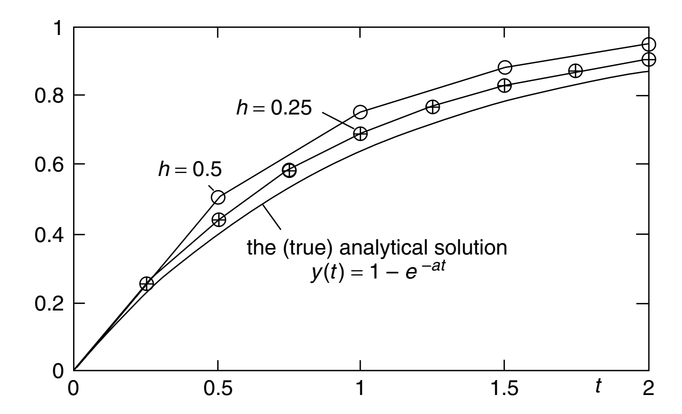


Figure 6.1 Examples of numerical solution obtained by using the Euler's method.

and is cast into the MATLAB routine "`ode_Euler()`".

```
function [t,y] = ode_Euler(f,tspan,y0,N)
%Euler's method to solve vector differential equation y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
if nargin<4 | N <= 0, N = 100; end
if nargin<3, y0 = 0; end
h = (tspan(2) - tspan(1))/N; %stepsize
t = tspan(1)+[0:N]'*h; %time vector
y(1,:) = y0(:)'; %always make the initial value a row vector
for k = 1:N
   y(k + 1,:) = y(k,:) +h*feval(f,t(k),y(k,:)); %Eq.(6.1.7)
end
```

6.2 HEUN'S METHOD: TRAPEZOIDAL METHOD

Another method of solving a first-order vector differential equation like Eq. (6.1.6) comes from integrating both sides of the equation.

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t)|_{t_k}^{t_{k+1}} = \mathbf{y}(t_{k+1}) - \mathbf{y}(t_k) = \int_{t_k}^{t_{k+1}} \mathbf{f}(t, \mathbf{y})\, dt$$

$$\mathbf{y}(t_{k+1}) = \mathbf{y}(t_k) + \int_{t_k}^{t_{k+1}} \mathbf{f}(t, \mathbf{y})\, dt \qquad \text{with } \mathbf{y}(t_0) = \mathbf{y}_0 \tag{6.2.1}$$

If we assume that the value of the (derivative) function $\mathbf{f}(t,\mathbf{y})$ is constant as $\mathbf{f}(t_k,\mathbf{y}(t_k))$ within one time step $[t_k,t_{k+1})$, this becomes Eq. (6.1.7) (with $h = t_{k+1} - t_k$), amounting to Euler's method. If we use the trapezoidal rule (5.5.3), it becomes

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{2}\{\mathbf{f}(t_k, \mathbf{y}_k) + \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1})\} \tag{6.2.2}$$

```
function [t,y] = ode_Heun(f,tspan,y0,N)
%Heun method to solve vector differential equation y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
if nargin<4 | N <= 0, N = 100; end
if nargin<3, y0 = 0; end
h = (tspan(2) - tspan(1))/N; %stepsize
t = tspan(1)+[0:N]'*h; %time vector
y(1,:) = y0(:)'; %always make the initial value a row vector
for k = 1:N
  fk = feval(f,t(k),y(k,:)); y(k+1,:) = y(k,:)+h*fk; %Eq.(6.2.3)
  y(k+1,:) = y(k,:) +h/2*(fk +feval(f,t(k+1),y(k+1,:))); %Eq.(6.2.4)
end
```

But, the right-hand side (RHS) of this equation has $\mathbf{y}_{k+1}$, which is unknown at t_k. To resolve this problem, we replace the $\mathbf{y}_{k+1}$ on the RHS by the following approximation:

$$\mathbf{y}_{k+1} \cong \mathbf{y}_k + h\mathbf{f}(t_k, \mathbf{y}_k) \tag{6.2.3}$$

so that it becomes

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{2}\{\mathbf{f}(t_k, \mathbf{y}_k) + \mathbf{f}(t_{k+1}, \mathbf{y}_k + h\mathbf{f}(t_k, \mathbf{y}_k))\} \tag{6.2.4}$$

This is Heun's method, which is implemented in the MATLAB routine "`ode_Heun()`". It is a kind of predictor-and-corrector method in that it predicts the value of $\mathbf{y}_{k+1}$ by Eq. (6.2.3) at t_k and then corrects the predicted value by Eq. (6.2.4) at t_{k+1}. The truncation error of Heun's method is $O(h^2)$ (proportional to h^2) as shown in Eq. (5.6.1), while the error of Euler's method is $O(h)$.

6.3 RUNGE–KUTTA METHOD

Although Heun's method is a little better than the Euler's method, it is still not accurate enough for most real-world problems. The fourth-order Runge–Kutta (RK4) method having a truncation error of $O(h^4)$ is one of the most widely used methods for solving differential equations. Its algorithm is described below.

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{6}(\mathbf{f}_{k1} + 2\mathbf{f}_{k2} + 2\mathbf{f}_{k3} + \mathbf{f}_{k4}) \tag{6.3.1}$$

where

$$\mathbf{f}_{k1} = \mathbf{f}(t_k, \mathbf{y}_k) \tag{6.3.2a}$$

$$\mathbf{f}_{k2} = \mathbf{f}(t_k + h/2, \mathbf{y}_k + \mathbf{f}_{k1}h/2) \tag{6.3.2b}$$

$$\mathbf{f}_{k3} = \mathbf{f}(t_k + h/2, \mathbf{y}_k + \mathbf{f}_{k2}h/2) \tag{6.3.2c}$$

$$\mathbf{f}_{k4} = \mathbf{f}(t_k + h, \mathbf{y}_k + \mathbf{f}_{k3}h) \tag{6.3.2d}$$

```
function [t,y] = ode_RK4(f,tspan,y0,N,varargin)
%Runge-Kutta method to solve vector differential eqn y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
if nargin < 4 | N <= 0, N = 100; end
if nargin < 3, y0 = 0; end
y(1,:) = y0(:)'; %make it a row vector
h = (tspan(2) - tspan(1))/N; t = tspan(1)+[0:N]'*h;
for k = 1:N
  f1 = h*feval(f,t(k),y(k,:),varargin{:}); f1 = f1(:)'; %(6.3.2a)
  f2 = h*feval(f,t(k) + h/2,y(k,:) + f1/2,varargin{:}); f2 = f2(:)';%(6.3.2b)
  f3 = h*feval(f,t(k) + h/2,y(k,:) + f2/2,varargin{:}); f3 = f3(:)';%(6.3.2c)
  f4 = h*feval(f,t(k) + h,y(k,:) + f3,varargin{:}); f4 = f4(:)'; %(6.3.2d)
  y(k + 1,:) = y(k,:) + (f1 + 2*(f2 + f3) + f4)/6; %Eq.(6.3.1)
end
```

```
%nm630: Heun/Euer/RK4 method to solve a differential equation (d.e.)
clear, clf
tspan = [0 2];
t = tspan(1)+[0:100]*(tspan(2) - tspan(1))/100;
a = 1; yt = 1 - exp(-a*t); %Eq.(6.1.5): true analytical solution
plot(t,yt,'k'), hold on
df61 = inline('-y + 1','t','y'); %Eq.(6.1.1): d.e. to be solved
y0 = 0;  N = 4;
[t1,ye] = oed_Euler(df61,tspan,y0,N);
[t1,yh] = ode_Heun(df61,tspan,y0,N);
[t1,yr] = ode_RK4(df61,tspan,y0,N);
plot(t,yt,'k', t1,ye,'b:', t1,yh,'b:', t1,yr,'r:')
plot(t1,ye,'bo', t1,yh,'b+', t1,yr,'r*')
N = 1e3; %to estimate the time for N iterations
tic, [t1,ye] = ode_Euler(df61,tspan,y0,N); time_Euler = toc
tic, [t1,yh] = ode_Heun(df61,tspan,y0,N); time_Heun = toc
tic, [t1,yr] = ode_RK4(df61,tspan,y0,N); time_RK4 = toc
```

Equation (6.3.1) is the core of RK4 method, which may be obtained by substituting Simpson's rule (5.5.4)

$$\int_{t_k}^{t_{k+1}} f(x)\,dx \cong \frac{h'}{3}(f_k + 4f_{k+1/2} + f_{k+1}) \qquad \text{with } h' = \frac{x_{k+1} - x_k}{2} = \frac{h}{2} \tag{6.3.3}$$

into the integral form (6.2.1) of differential equation and replacing $f_{k+1/2}$ with the average of the successive function values $(f_{k2} + f_{k3})/2$. Accordingly, the RK4 method has a truncation error of $O(h^4)$ as Eq. (5.6.2) and thus is expected to work better than the previous two methods.

The fourth-order Runge–Kutta (RK4) method is cast into the MATLAB routine "`ode_RK4()`". The program "`nm630.m`" uses this routine to solve Eq. (6.1.1) with the step size $h = (t_f - t_0)/N = 2/4 = 0.5$ and plots the numerical result together with the (true) analytical solution. Comparison of this result with those of Euler's method ("`ode_Euler()`") and Heun's method ("`ode_Heun()`") is given in Fig. 6.2, which shows that the RK4 method is better than Heun's method, while Euler's method is the worst in terms of accuracy with the same step-size. But,

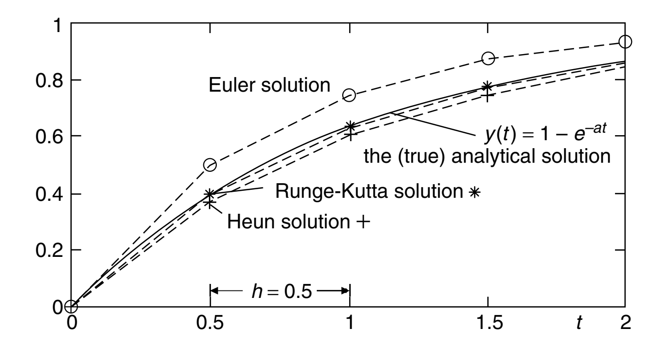


Figure 6.2 Numerical solutions for a first-order differential equation.

in terms of computational load, the order is reversed, because Euler's method, Heun's method, and the RK4 method need 1, 2, and 4 function evaluations (calls) per iteration, respectively.

(cf) Note that a function call takes much more time than a multiplication and thus the number of function calls should be a criterion in estimating and comparing computational time.

The MATLAB built-in routines "`ode23()`" and "`ode45()`" implement the Runge–Kutta method with an adaptive step-size adjustment, which uses a large/small step-size depending on whether $f(t)$ is smooth or rough. In Section 6.4.3, we will try applying these routines together with our routines to solve a differential equation for practice rather than for comparison.

6.4 PREDICTOR–CORRECTOR METHOD

6.4.1 Adams–Bashforth–Moulton Method

The Adams–Bashforth–Moulton (ABM) method consists of two steps. The first step is to approximate $\mathbf{f}(t,\mathbf{y})$ by the (Lagrange) polynomial of degree 4 matching the four points

$$\{(t_{k-3}, \mathbf{f}_{k-3}), (t_{k-2}, \mathbf{f}_{k-2}), (t_{k-1}, \mathbf{f}_{k-1}), (t_k, \mathbf{f}_k)\}$$

and substitute the polynomial into the integral form (6.2.1) of differential equation to get a predicted estimate of $\mathbf{y}_{k+1}$.

$$\mathbf{p}_{k+1} = \mathbf{y}_k + \int_0^h l_3(t)\,dt = \mathbf{y}_k + \frac{h}{24}(-9\mathbf{f}_{k-3} + 37\mathbf{f}_{k-2} - 59\mathbf{f}_{k-1} + 55\mathbf{f}_k) \tag{6.4.1a}$$

The second step is to repeat the same work with the updated four points

$$\{(t_{k-2}, \mathbf{f}_{k-2}), (t_{k-1}, \mathbf{f}_{k-1}), (t_k, \mathbf{f}_k), (t_{k+1}, \mathbf{f}_{k+1})\} \quad (\mathbf{f}_{k+1} = \mathbf{f}(t_{k+1}, \mathbf{p}_{k+1}))$$

to get a corrected estimate of $\mathbf{y}_{k+1}$.

$$\mathbf{c}_{k+1} = \mathbf{y}_k + \int_0^h l_3'(t)\,dt = \mathbf{y}_k + \frac{h}{24}(\mathbf{f}_{k-2} - 5\mathbf{f}_{k-1} + 19\mathbf{f}_k + 9\mathbf{f}_{k+1}) \tag{6.4.1b}$$

The coefficients of Eqs. (6.4.1a) and (6.4.1b) can be obtained by using the MATLAB routines "`lagranp()`" and "`polyint()`", each of which generates Lagrange (coefficient) polynomials and integrates a polynomial, respectively. Let's try running the program "`ABMc.m`".

```
>>abmc
   cAP  = -3/8    37/24  -59/24   55/24
   cAC  =  1/24   -5/24   19/24    3/8
```

```
%ABMc.m
% Predictor/Corrector coefficients in Adams-Bashforth-Moulton method
clear
format rat
[l,L] = lagranp([-3 -2 -1 0],[0 0 0 0]); %only coefficient polynomial L
for m = 1:4
   iL = polyint(L(m,:)); %indefinite integral of polynomial
   cAP(m) = polyval(iL,1)-polyval(iL,0); %definite integral over [0,1]
end
cAP %Predictor coefficients
[l,L] = lagranp([-2 -1 0 1],[0 0 0 0]); %only coefficient polynomial L
for m = 1:4
   iL = polyint(L(m,:)); %indefinite integral of polynomial
   cAC(m) = polyval(iL,1) - polyval(iL,0); %definite integral over [0,1]
end
cAC %Corrector coefficients
format short
```

Alternatively, we write the Taylor series expansion of $\mathbf{y}_{k+1}$ about t_k and that of $\mathbf{y}_k$ about t_{k+1} as

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}_k + \frac{h^2}{2}\mathbf{f}_k' + \frac{h^3}{3!}\mathbf{f}_k^{(2)} + \frac{h^4}{4!}\mathbf{f}_k^{(3)} + \frac{h^5}{5!}\mathbf{f}_k^{(4)} + \cdots \tag{6.4.2a}$$

$$\mathbf{y}_k = \mathbf{y}_{k+1} - h\mathbf{f}_{k+1} + \frac{h^2}{2}\mathbf{f}_{k+1}' - \frac{h^3}{3!}\mathbf{f}_{k+1}^{(2)} + \frac{h^4}{4!}\mathbf{f}_{k+1}^{(3)} - \frac{h^5}{5!}\mathbf{f}_{k+1}^{(4)} + \cdots$$

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}_{k+1} - \frac{h^2}{2}\mathbf{f}_{k+1}' + \frac{h^3}{3!}\mathbf{f}_{k+1}^{(2)} - \frac{h^4}{4!}\mathbf{f}_{k+1}^{(3)} + \frac{h^5}{5!}\mathbf{f}_{k+1}^{(4)} - \cdots \tag{6.4.2b}$$

and replace the first, second, and third derivatives by their difference approximations.

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}_k + \frac{h^2}{2}\left(\frac{-\frac{1}{3}\mathbf{f}_{k-3} + \frac{3}{2}\mathbf{f}_{k-2} - 3\mathbf{f}_{k-1} + \frac{11}{6}\mathbf{f}_k}{h} + \frac{1}{4}h^3\mathbf{f}_k^{(4)} + \cdots\right)$$
$$+ \frac{h^3}{3!}\left(\frac{-\mathbf{f}_{k-3} + 4\mathbf{f}_{k-2} - 5\mathbf{f}_{k-1} + 2\mathbf{f}_k}{h^2} + \frac{11}{12}h^2\mathbf{f}_k^{(4)} + \cdots\right)$$
$$+ \frac{h^4}{4!}\left(\frac{-\mathbf{f}_{k-3} + 3\mathbf{f}_{k-2} - 3\mathbf{f}_{k-1} + \mathbf{f}_k}{h^3} + \frac{3}{2}h\mathbf{f}_k^{(4)} + \cdots\right) + \frac{h^5}{120}\mathbf{f}_k^{(4)} + \cdots$$
$$= \mathbf{y}_k + \frac{h}{24}(-9\mathbf{f}_{k-3} + 37\mathbf{f}_{k-2} - 59\mathbf{f}_{k-1} + 55\mathbf{f}_k) + \frac{251}{720}h^5\mathbf{f}_k^{(4)} + \cdots$$
$$\overset{(6.4.1a)}{\approx} \mathbf{p}_{k+1} + \frac{251}{720}h^5\mathbf{f}_k^{(4)} \tag{6.4.3a}$$

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}_{k+1} - \frac{h^2}{2}\left(\frac{-\frac{1}{3}\mathbf{f}_{k-2} + \frac{3}{2}\mathbf{f}_{k-1} - 3\mathbf{f}_k + \frac{11}{6}\mathbf{f}_{k+1}}{h} + \frac{1}{4}h^3\mathbf{f}_{k+1}^{(4)} + \cdots\right)$$
$$+ \frac{h^3}{3!}\left(\frac{-\mathbf{f}_{k-2} + 4\mathbf{f}_{k-1} - 5\mathbf{f}_k + 2\mathbf{f}_{k+1}}{h^2} + \frac{11}{12}h^2\mathbf{f}_{k+1}^{(4)} + \cdots\right)$$
$$- \frac{h^4}{4!}\left(\frac{-\mathbf{f}_{k-2} + 3\mathbf{f}_{k-1} - 3\mathbf{f}_k + \mathbf{f}_{k+1}}{h^3} + \frac{3}{2}h\mathbf{f}_{k+1}^{(4)} + \cdots\right) + \frac{h^5}{120}\mathbf{f}_{k+1}^{(4)} + \cdots$$
$$= \mathbf{y}_k + \frac{h}{24}(\mathbf{f}_{k-2} - 5\mathbf{f}_{k-1} + 19\mathbf{f}_k + 9\mathbf{f}_{k+1}) - \frac{19}{720}h^5\mathbf{f}_{k+1}^{(4)} + \cdots$$
$$\overset{(6.4.1b)}{\approx} \mathbf{c}_{k+1} - \frac{19}{720}h^5\mathbf{f}_{k+1}^{(4)} \tag{6.4.3b}$$

These derivations are supported by running the MATLAB program “ABMc1.m”.

```
%ABMc1.m
%another way to get the ABM coefficients together with the error term
clear, format rat
for i = 1:3, [ci,erri] = difapx(i,[-3 0]); c(i,:) = ci; err(i) = erri;
    end
cAP = [0 0 0 1]+[1/2 1/6 1/24]*c, errp = -[1/2 1/6 1/24]*err' + 1/120
cAC = [0 0 0 1]+[-1/2 1/6 -1/24]*c, errc = -[-1/2 1/6 -1/24]*err' + 1/120
format short
```

From these equations and under the assumption that $\mathbf{f}_{k+1}^{(4)} \cong \mathbf{f}_k^{(4)} \cong K$, we can write the predictor/corrector errors as

$$E_{P,k+1} = \mathbf{y}_{k+1} - \mathbf{p}_{k+1} \approx \frac{251}{720}h^5\mathbf{f}_k^{(4)} \cong \frac{251}{720}Kh^5 \tag{6.4.4a}$$

$$E_{C,k+1} = \mathbf{y}_{k+1} - \mathbf{c}_{k+1} \approx -\frac{19}{720}h^5\mathbf{f}_{k+1}^{(4)} \cong -\frac{19}{720}Kh^5 \tag{6.4.4b}$$

We still cannot use these formulas to estimate the predictor/corrector errors, since K is unknown. But, from the difference between these two formulas

$$E_{P,k+1} - E_{C,k+1} = \mathbf{c}_{k+1} - \mathbf{p}_{k+1} \cong \frac{270}{720}Kh^5 \equiv \frac{270}{251}E_{P,k+1} \equiv -\frac{270}{19}E_{C,k+1} \tag{6.4.5}$$

we can get the practical formulas for estimating the errors as

$$E_{P,k+1} = \mathbf{y}_{k+1} - \mathbf{p}_{k+1} \cong \frac{251}{270}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \tag{6.4.6a}$$

$$E_{C,k+1} = \mathbf{y}_{k+1} - \mathbf{c}_{k+1} \cong -\frac{19}{270}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \tag{6.4.6b}$$

These formulas give us rough estimates of how close the predicted/corrected values are to the true value and so can be used to improve them as well as to adjust the step-size.

$$\mathbf{p}_{k+1} \to \mathbf{p}_{k+1} + \frac{251}{270}(\mathbf{c}_k - \mathbf{p}_k) \Rightarrow \mathbf{m}_{k+1} \tag{6.4.7a}$$

$$\mathbf{c}_{k+1} \to \mathbf{c}_{k+1} - \frac{19}{270}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \Rightarrow \mathbf{y}_{k+1} \tag{6.4.7b}$$

These modification formulas are expected to reward our efforts that we have made to derive them.

The Adams–Bashforth–Moulton (ABM) method with the modification formulas can be described by Eqs. (6.4.1a), (6.4.1b), and (6.4.7a), (6.4.7b) summarized below and is cast into the MATLAB routine "`ode_ABM()`". This scheme needs only two function evaluations (calls) per iteration, while having a truncation error of $O(h^5)$ and thus is expected to work better than the methods discussed so far. It is implemented by the MATLAB built-in routine "`ode113()`" with many additional sophisticated techniques.

⟨Adams–Bashforth–Moulton method with modification formulas⟩

Predictor: $$\mathbf{p}_{k+1} = \mathbf{y}_k + \frac{h}{24}(-9\mathbf{f}_{k-3} + 37\mathbf{f}_{k-2} - 59\mathbf{f}_{k-1} + 55\mathbf{f}_k) \tag{6.4.8a}$$

Modifier: $$\mathbf{m}_{k+1} = \mathbf{p}_{k+1} + \frac{251}{270}(\mathbf{c}_k - \mathbf{p}_k) \tag{6.4.8b}$$

Corrector: $$\mathbf{c}_{k+1} = \mathbf{y}_k + \frac{h}{24}(\mathbf{f}_{k-2} - 5\mathbf{f}_{k-1} + 19\mathbf{f}_k + 9\mathbf{f}(t_{k+1}, \mathbf{m}_{k+1})) \tag{6.4.8c}$$

$$\mathbf{y}_{k+1} = \mathbf{c}_{k+1} - \frac{19}{270}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \tag{6.4.8d}$$

```
function [t,y] = ode_ABM(f,tspan,y0,N,KC,varargin)
%Adams-Bashforth-Moulton method to solve vector d.e. y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
% using the modifier based on the error estimate depending on KC = 1/0
if nargin < 5, KC = 1; end %with modifier by default
if nargin < 4 | N < = 0, N = 100; end %default maximum number of iterations
y0 = y0(:)'; %make it a row vector
h = (tspan(2) - tspan(1))/N; %step size
tspan0 = tspan(1)+[0 3]*h;
[t,y] = rk4(f,tspan0,y0,3,varargin{:}); %initialize by Runge-Kutta
t = [t(1:3)'  t(4):h:tspan(2)]';
for k = 1:4, F(k,:) = feval(f,t(k),y(k,:),varargin{:}); end
p = y(4,:); c = y(4,:); KC22 = KC*251/270; KC12 = KC*19/270;
h24 = h/24; h241 = h24*[1 -5 19 9]; h249 = h24*[-9 37 -59 55];
for k = 4:N
  p1 = y(k,:) +h249*F; %Eq.(6.4.8a)
  m1 = pk1 + KC22*(c-p); %Eq.(6.4.8b)
  c1 = y(k,:)+  ...
        h241*[F(2:4,:); feval(f,t(k + 1),m1,varargin{:})]; %Eq.(6.4.8c)
  y(k + 1,:) = c1 - KC12*(c1 - p1); %Eq.(6.4.8d)
  p = p1;  c = c1; %update the predicted/corrected values
  F = [F(2:4,:); feval(f,t(k + 1),y(k + 1,:),varargin{:})];
End
```

6.4.2 Hamming Method

```
function [t,y] = ode_Ham(f,tspan,y0,N,KC,varargin)
% Hamming method to solve vector d.e. y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
% using the modifier based on the error estimate depending on KC = 1/0
if nargin < 5, KC = 1; end %with modifier by default
if nargin < 4 | N <= 0, N = 100; end %default maximum number of iterations
if nargin < 3, y0 = 0; end %default initial value
y0 = y0(:)'; end %make it a row vector
h = (tspan(2)-tspan(1))/N; %step size
tspan0 = tspan(1)+[0 3]*h;
[t,y] = ode_RK4(f,tspan0,y0,3,varargin{:}); %Initialize by Runge-Kutta
t = [t(1:3)'  t(4):h:tspan(2)]';
for k = 2:4, F(k - 1,:) = feval(f,t(k),y(k,:),varargin{:}); end
p = y(4,:); c = y(4,:); h34 = h/3*4; KC11 = KC*112/121; KC91 = KC*9/121;
h312 = 3*h*[-1 2 1];
for k = 4:N
   p1 = y(k - 3,:) + h34*(2*(F(1,:) + F(3,:)) - F(2,:)); %Eq.(6.4.9a)
   m1 = p1 + KC11*(c - p);               %Eq.(6.4.9b)
   c1 = (-y(k - 2,:) + 9*y(k,:) +...
       h312*[F(2:3,:); feval(f,t(k + 1),m1,varargin{:})])/8; %Eq.(6.4.9c)
   y(k+1,:) = c1 - KC91*(c1 - p1);               %Eq.(6.4.9d)
   p = p1;  c = c1;  %update the predicted/corrected values
   F = [F(2:3,:); feval(f,t(k + 1),y(k + 1,:),varargin{:})];
end
```

⟨Hamming method with modification formulas⟩

Predictor: $$\mathbf{p}_{k+1} = \mathbf{y}_{k-3} + \frac{4h}{3}(2\mathbf{f}_{k-2} - \mathbf{f}_{k-1} + 2\mathbf{f}_k) \quad (6.4.9a)$$

Modifier: $$\mathbf{m}_{k+1} = \mathbf{p}_{k+1} + \frac{112}{121}(\mathbf{c}_k - \mathbf{p}_k) \quad (6.4.9b)$$

Corrector: $$\mathbf{c}_{k+1} = \frac{1}{8}\{9\mathbf{y}_k - \mathbf{y}_{k-2} + 3h(-\mathbf{f}_{k-1} + 2\mathbf{f}_k + \mathbf{f}(t_{k+1}, \mathbf{m}_{k+1}))\} \quad (6.4.9c)$$

$$\mathbf{y}_{k+1} = \mathbf{c}_{k+1} - \frac{9}{121}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \quad (6.4.9d)$$

In this section, we introduce just the algorithm of the Hamming method [H-1] summarized in the box above and the corresponding routine “`ode_Ham()`”, which is another multistep predictor–corrector method like the Adams–Bashforth–Moulton (ABM) method.

This scheme also needs only two function evaluations (calls) per iteration, while having the error of $O(h^5)$ and so is comparable with the ABM method discussed in the previous section.

6.4.3 Comparison of Methods

The major factors to be considered in evaluating/comparing different numerical methods are the accuracy of the numerical solution and its computation time. In this section, we will compare the routines “`ode_RK4()`”, “`ode_ABM()`”, “`ode_Ham()`”, “`ode23()`”, “`ode45()`”, and “`ode113()`” by trying them out on the same differential equations, hopefully to make some conjectures about their performances. It is important to note that the evaluation/comparison of numerical methods is not so simple because their performances may depend on the characteristic of the problem at hand. It should also be noted that there are other factors to be considered, such as stability, versatility, proof against run-time error, and so on. These points are being considered in most of the MATLAB built-in routines.

The first thing we are going to do is to validate the effectiveness of the modifiers (Eqs. (6.4.8b,d) and (6.4.9b,d)) in the ABM (Adams–Bashforth–Moulton) method and the Hamming method. For this job, we write and run the program “`nm643_1.m`” to get the results depicted in Fig. 6.3 for the differential equation

$$y'(t) = -y(t) + 1 \qquad \text{with } y(0) = 0 \quad (6.4.10)$$

which was given at the beginning of this chapter. Fig. 6.3 shows us an interesting fact that, although the ABM method and the Hamming method, even without modifiers, are theoretically expected to have better accuracy than the RK4 (fourth-order Runge–Kutta) method, they turn out to work better than RK4 only with modifiers. Of course, it is not always the case, as illustrated in Fig. 6.4, which

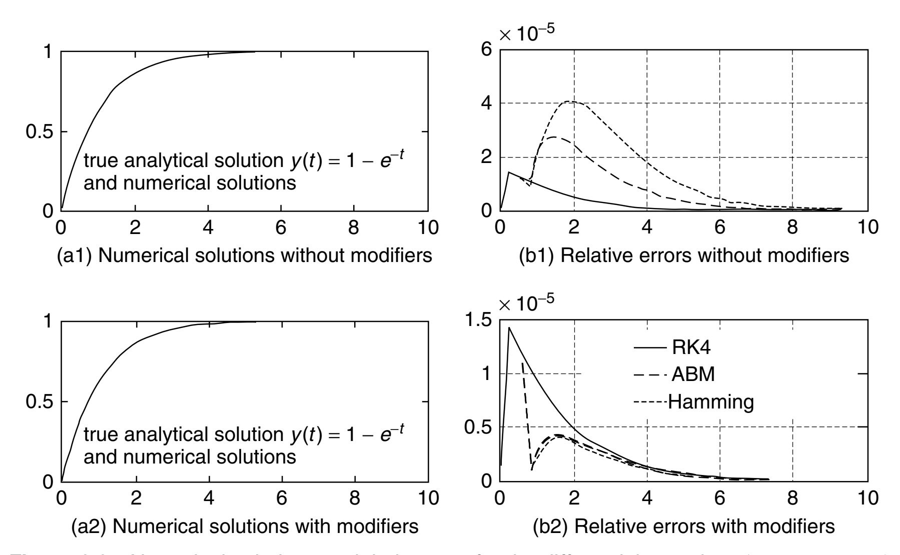


Figure 6.3 Numerical solutions and their errors for the differential equation $y'(t) = -y(t) + 1$.

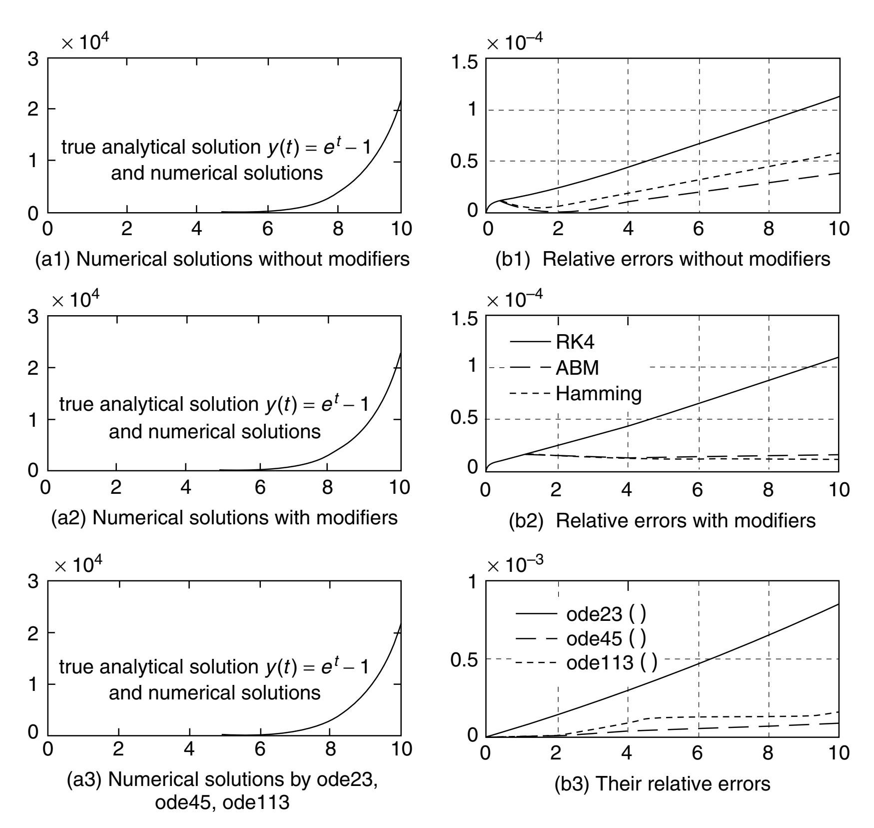


Figure 6.4 Numerical solutions and their errors for the differential equation $y'(t) = y(t) + 1$.

we obtained by applying the same routines to solve another differential equation

$$y'(t) = y(t) + 1 \qquad \text{with } y(0) = 0 \tag{6.4.11}$$

where the true analytical solution is

$$y(t) = e^t - 1 \tag{6.4.12}$$

```
%nm643_1: RK4/Adams/Hamming method to solve a differential eq
clear, clf
t0 = 0; tf = 10; y0 = 0; %starting/final time, initial value
N = 50; %number of segments
df643 = inline('-y+1','t','y'); %differential equation to solve
f643 = inline('1-exp(-t)','t'); %true analytical solution
for KC = 0:1
   tic, [t1,yR] = ode_RK4(df643,[t0 tf],y0,N); tR = toc
   tic, [t1,yA] = ode_ABM(df643,[t0 tf],y0,N,KC); tA = toc
   tic, [t1,yH] = ode_Ham(df643,[t0 tf],y0,N,KC); tH = toc
   yt1 = f643(t1); %true analytical solution to plot
   subplot(221 + KC*2) %plot analytical/numerical solutions
   plot(t1,yt1,'k', t1,yR,'k', t1,yA,'k--', t1,yH,'k:')
   tmp = abs(yt1)+eps; l_t1 = length(t1);
   eR = abs(yR - yt1)./tmp; e_R=norm(eR)/lt1
   eA = abs(yA - yt1)./tmp; e_A=norm(eA)/lt1
   eH = abs(yH - yt1)./tmp; e_H=norm(eH)/lt1
   subplot(222 + KC*2) %plot relative errors
   plot(t1,eR,'k', t1,eA,'k--', t1, eH,'k:')
end
```

```
%nm643_2: ode23()/ode45()/ode113() to solve a differential eq
clear, clf
t0 = 0; tf = 10; y0 = 0; N = 50; %starting/final time, initial value
df643 = inline('y + 1','t','y'); %differential equation to solve
f643 = inline('exp(t) - 1','t'); %true analytical solution
tic, [t1,yR] = ode_RK4(df643,[t0 tf],y0,N); time(1) = toc;
tic, [t1,yA] = ode_ABM(df643,[t0 tf],y0,N); time(2) = toc;
yt1 = f643(t1);
tmp = abs(yt1)+ eps; l_t1 = length(t1);
eR = abs(yR-yt1)./tmp; err(1) = norm(eR)/l_t1;
eA = abs(yA-yt1)./tmp; err(2) = norm(eA)/l_t1;
options = odeset('RelTol',1e-4); %set the tolerance of relative error
tic, [t23,yode23] = ode23(df643,[t0 tf],y0,options); time(3) = toc;
tic, [t45,yode45] = ode45(df643,[t0 tf],y0,options); time(4) = toc;
tic, [t113,yode113] = ode113(df643,[t0 tf],y0,options); time(5) = toc;
yt23 = f643(t23); tmp = abs(yt23) + eps;
eode23 = abs(yode23-yt23)./tmp; err(3) = norm(eode23)/length(t23);
yt45 = f643(t45); tmp = abs(yt45) + eps;
eode45 = abs(yode45 - yt45)./tmp; err(4) = norm(eode45)/length(t45);
yt113 = f643(t113); tmp = abs(yt113) + eps;
eode113 = abs(yode113 - yt113)./tmp; err(5) = norm(eode113)/length(t113);
subplot(221), plot(t23,yode23,'k', t45,yode45,'b', t113,yode113,'r')
subplot(222), plot(t23,eode23,'k', t45,eode45,'b--', t113,eode113,'r:')
err, time
```

Table 6.2 Results of Applying Several Routines to solve a Simple Differential Equation

	`ode_RK4()`	`ode_ABM()`	`ode_Ham()`	`ode23()`	`ode45()`	`ode113()`
Relative error	0.0925×10^{-4}	0.0203×10^{-4}	0.0179×10^{-4}	0.4770×10^{-4}	0.0422×10^{-4}	0.1249×10^{-4}
Computing time	0.05 sec	0.03 sec	0.03 sec	0.07 sec	0.05 sec	0.05 sec

Readers are invited to supplement the program "`nm643_2.m`" in such a way that "`ode_Ham()`" is also used to solve Eq. (6.4.11). Running the program yields the results depicted in Fig. 6.4 and listed in Table 6.2. From Fig. 6.4, it is noteworthy that, without the modifiers, the ABM method seems to be better than the Hamming method; however, with the modifiers, it is the other way around or at least they run a neck-and-neck race. Anyone will see that the predictor–corrector methods such as the ABM method (`ode_ABM()`) and the Hamming method (`ode_Ham()`) give us a better numerical solution with less error and shorter computation time than the MATLAB built-in routines "`ode23()`", "`ode45()`", and "`ode113()`" as well as the RK4 method (`ode_RK4()`), as listed in Table 6.2. But, a general conclusion should not be deduced just from one example.

6.5 VECTOR DIFFERENTIAL EQUATIONS

6.5.1 State Equation

Although we have tried using the MATLAB routines only for scalar differential equations, all the routines made by us or built inside MATLAB are ready to entertain first-order vector differential equations, called state equations, as below.

$$x_1{}'(t) = f_1(t, x_1(t), x_2(t), \ldots) \qquad \text{with } x_1(t_0) = x_{10}$$
$$x_2{}'(t) = f_2(t, x_1(t), x_2(t), \ldots) \qquad \text{with } x_2(t_0) = x_{20}$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$
$$\mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x}(t)) \qquad \text{with } \mathbf{x}(t_0) = \mathbf{x}_0 \tag{6.5.1}$$

For example, we can define the system of first-order differential equations

$$\begin{aligned} x_1{}'(t) &= x_2(t) && \text{with } x_1(0) = 1 \\ x_2{}'(t) &= -x_2(t) + 1 && \text{with } x_2(0) = -1 \end{aligned} \tag{6.5.2}$$

in a file named "`df651.m`" and solve it by running the MATLAB program "`nm651_1.m`", which uses the routines "`ode_Ham()`"/"`ode45()`" to get the numerical solutions and plots the results as depicted in Fig. 6.5. Note that the function given as the first input argument of "`ode45()`" must be fabricated to generate its value in a column vector or at least, in the same form of vector as the input argument '`x`' so long as it is a vector-valued function.

```
%nm651_1 to solve a system of differential eqs., i.e., state equation
df = 'df651';
t0 = 0; tf = 2; x0 = [1 -1]; %start/final time and initial value
N = 45; [tH,xH] = ode_Ham(df,[t0 tf],x0,N); %with N = number of segments
[t45,x45] = ode45(df,[t0 tf],x0);
plot(tH,xH), hold on, pause, plot(t45,x45)
```

```
function dx = df651(t,x)
dx = zeros(size(x)); %row/column vector depending on the shape of x
dx(1) = x(2);  dx(2) = -x(2) + 1;
```

Especially for the state equations having only constant coefficients like Eq. (6.5.2), we can change it into a matrix–vector form as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_s(t) \tag{6.5.3}$$

$$\text{with } \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ and } u_s(t) = 1 \ \forall \ t \geq 0$$

$$\mathbf{x}'(t) = A\mathbf{x}(t) + Bu(t) \text{ with the initial state } \mathbf{x}(0) \text{ and the input } u(t) \tag{6.5.4}$$

which is called a linear time-invariant (LTI) state equation, and then try to find the analytical solution. For this purpose, we take the Laplace transform of both sides to write

$$sX(s) - \mathbf{x}(0) = AX(s) + BU(s) \quad \text{with } X(s) = L\{x(t)\}, U(s) = L\{u(t)\}$$

$$[sI - A]X(s) = \mathbf{x}(0) + BU(s), \quad X(s) = [sI - A]^{-1}\mathbf{x}(0) + [sI - A]^{-1}BU(s) \tag{6.5.5}$$

where $L\{x(t)\}$ and $L^{-1}\{X(s)\}$ denote the Laplace transform of $x(t)$ and the inverse Laplace transform of $X(s)$, respectively. Note that

$$[sI - A]^{-1} = s^{-1}[I - As^{-1}]^{-1} = s^{-1}\left\lfloor I + As^{-1} + A^2s^{-2} + \cdots \right\rfloor$$

$$\phi(t) = L^{-1}\{[sI - A]^{-1}\} \tag{6.5.6}$$

$$= I + At + \frac{A^2}{2}t^2 + \frac{A^3}{3!}t^3 + \cdots = e^{At} \quad \text{with } \phi(0) = I$$

By applying the convolution property of Laplace transform (Table D.2(4) in Appendix D)

$$L^{-1}\{[sI - A]^{-1}BU(s)\} = L^{-1}\{[sI - A]^{-1}\} * L^{-1}\{BU(s)\} = \phi(t) * Bu(t)$$

$$= \int_{-\infty}^{\infty} \phi(t - \tau)Bu(\tau)\,d\tau \overset{u(\tau)=0 \text{ for } \tau<0 \text{ or } \tau>t}{=} \int_0^t \phi(t - \tau)Bu(\tau)\,d\tau \tag{6.5.7}$$

we can take the inverse Laplace transform of Eq. (6.5.5) to write

$$\mathbf{x}(t) = \phi(t)\mathbf{x}(0) + \phi(t) * Bu(t) = \phi(t)\mathbf{x}(0) + \int_0^t \phi(t-\tau)Bu(\tau)\,d\tau \quad (6.5.8)$$

For Eq. (6.5.3), we use Eq. (6.5.6) to find

$$\begin{aligned}\phi(t) &= L^{-1}\{[sI - A]^{-1}\} \\ &= L^{-1}\left\{\left[\begin{bmatrix} s & 0 \\ 0 & s\end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 0 & -1\end{bmatrix}\right]^{-1}\right\} = L^{-1}\left\{\begin{bmatrix} s & -1 \\ 0 & s+1\end{bmatrix}^{-1}\right\} \\ &= L^{-1}\left\{\frac{1}{s(s+1)}\begin{bmatrix} s+1 & 1 \\ 0 & s\end{bmatrix}\right\} \\ &= L^{-1}\left\{\begin{bmatrix} 1/s & 1/s - 1/(s+1) \\ 0 & 1/(s+1)\end{bmatrix}\right\} = \begin{bmatrix} 1 & 1-e^{-t} \\ 0 & e^{-t}\end{bmatrix} \end{aligned} \quad (6.5.9)$$

and use Eqs. (6.5.8), (6.5.9), and $u(t) = u_s(t) = 1 \ \forall\ t \geq 0$ to obtain

$$\begin{aligned}\mathbf{x}(t) &= \begin{bmatrix} 1 & 1-e^{-t} \\ 0 & e^{-t}\end{bmatrix}\begin{bmatrix} 1 \\ -1\end{bmatrix} + \int_0^t \begin{bmatrix} 1 & 1-e^{-(t-\tau)} \\ 0 & e^{-(t-\tau)}\end{bmatrix}\begin{bmatrix} 0 \\ 1\end{bmatrix} 1\,d\tau \\ &= \begin{bmatrix} e^{-t} \\ -e^{-t}\end{bmatrix} + \begin{bmatrix} \tau - e^{-(t-\tau)} \\ e^{-(t-\tau)}\end{bmatrix}\Bigg|_0^t = \begin{bmatrix} t-1+2e^{-t} \\ 1-2e^{-t}\end{bmatrix}\end{aligned} \quad (6.5.10)$$

Alternatively, we can directly take the inverse transform of Eq. (6.5.5) to get

$$\begin{aligned}X(s) &= [sI - A]^{-1}\{\mathbf{x}(0) + [sI - A]^{-1}BU(s)\} \\ &= \frac{1}{s(s+1)}\begin{bmatrix} s+1 & 1 \\ 0 & s\end{bmatrix}\left\{\begin{bmatrix} 1 \\ -1\end{bmatrix} + \begin{bmatrix} 0 \\ 1\end{bmatrix}\frac{1}{s}\right\} \\ &= \frac{1}{s^2(s+1)}\begin{bmatrix} s+1 & 1 \\ 0 & s\end{bmatrix}\begin{bmatrix} s \\ -s+1\end{bmatrix} = \frac{1}{s^2(s+1)}\begin{bmatrix} s^2+1 \\ s(1-s)\end{bmatrix}\end{aligned} \quad (6.5.11)$$

$$X_1(s) = \frac{s^2+1}{s^2(s+1)} = \frac{1}{s^2} - \frac{1}{s} + \frac{2}{s+1}, \quad x_1(t) = t - 1 + 2e^{-t} \quad (6.5.12a)$$

$$X_2(s) = \frac{1-s}{s(s+1)} = \frac{1}{s} - \frac{2}{s+1}, \quad x_2(t) = 1 - 2e^{-t} \quad (6.5.12b)$$

which conforms with Eq. (6.5.10).

The MATLAB program "`nm651_2.m`" uses a symbolic computation routine "`ilaplace()`" to get the inverse Laplace transform, uses "`eval()`" to evaluate

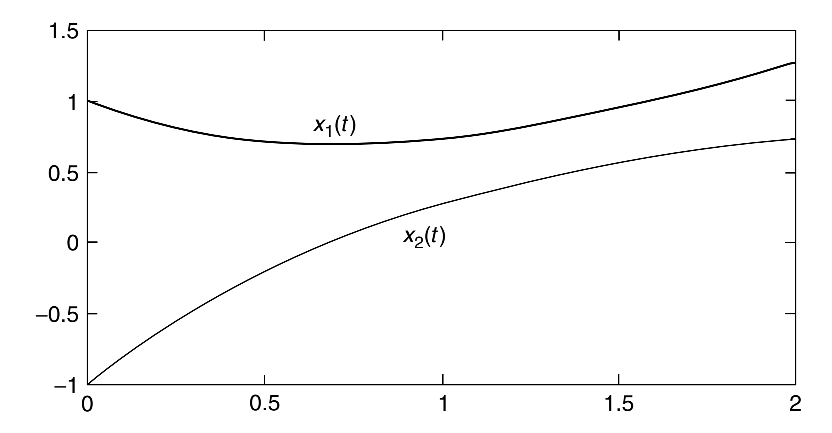


Figure 6.5 Numerical/analytical solutions of the continuous-time state equation (6.5.2)/(6.5.3).

it, and plots the result as depicted in Fig. 6.5, which supports this derivation procedure. Additionally, it uses another symbolic computation routine "`dsolve()`" to get the analytical solution directly.

```
>>nm651_2
   Solution of Differential Equation based on Laplace transform
    Xs  = [ 1/s + 1/s/(s + 1)*(-1 + 1/s) ]
          [       1/(s + 1)*(-1 + 1/s)   ]
    xt  = [ -1 + t + 2*exp(-t) ]
          [   -2*exp(-t) + 1   ]

   Analytical solution
     xt1  =  -1 + t + 2*exp(-t)
     xt2  =  -2*exp(-t) + 1
```

```
%nm651_2: Analytical solution for state eq. x'(t) = Ax(t) + Bu(t)(6.5.3)
clear
syms s t %declare s,t as symbolic variables
A = [0 1;0 -1]; B = [0 1]'; %Eq.(6.5.3)
x0 = [1 -1]'; %initial value
disp('Solution of Differential Eq based on Laplace transform')
disp('Laplace transformed solution X(s)')
Xs = (s*eye(size(A)) - A)^-1*(x0 + B/s) %Eq.(6.5.5)
disp('Inverse Laplace transformed solution x(t)')
xt = ilaplace(Xs) %inverse Laplace transform %Eq.(6.5.12)
t0 = 0; tf = 2; N = 45; %initial/final time
t = t0 + [0:N]'*(tf - t0)/N; %time vector
xtt = eval(xt:); %evaluate the inverse Laplace transform
plot(t,xtt)
disp('Analytical solution')
xt = dsolve('Dx1 = x2, Dx2 = -x2 + 1', 'x1(0) = 1, x2(0) = -1');
xt1 = xt.x1, xt2 = xt.x2 %Eq.(6.5.10)
```

6.5.2 Discretization of LTI State Equation

In this section, we consider a discretization method of converting a continuous-time LTI (linear time-invariant) state equation

$$\mathbf{x}'(t) = A\mathbf{x}(t) + Bu(t) \quad \text{with the initial state } \mathbf{x}(0) \text{ and the input } u(t) \tag{6.5.13}$$

into an equivalent discrete-time LTI state equation with the sampling period T

$$\mathbf{x}[n+1] = A_d\mathbf{x}[n] + B_d u[n] \tag{6.5.14}$$

with the initial state $\mathbf{x}[0]$ and the input $u[n] = u(nT)$ for $nT \le t < (n+1)T$

which can be solved easily by an iterative scheme mobilizing just simple multiplications and additions.

For this purpose, we rewrite the solution (6.5.8) of the continuous-time LTI state equation with the initial time t_0 as

$$\mathbf{x}(t) = \phi(t - t_0)\mathbf{x}(t_0) + \int_{t_0}^{t} \phi(t-\tau)Bu(\tau)\,d\tau \tag{6.5.15}$$

Under the assumption that the input is constant as the initial value within each sampling interval—that is, $u[n] = u(nT)$ for $nT \le t < (n+1)T$—we substitute $t_0 = nT$ and $t = (n+1)T$ into this equation to write the discrete-time LTI state equation as

$$\mathbf{x}((n+1)T) = \phi(T)\mathbf{x}(nT) + \int_{nT}^{(n+1)T} \phi((n+1)T - \tau)Bu(nT)\,d\tau$$

$$\mathbf{x}[n+1] = \phi(T)\mathbf{x}[n] + \int_{nT}^{(n+1)T} \phi(nT + T - \tau)\,d\tau\,Bu[n]$$

$$\mathbf{x}[n+1] = A_d\mathbf{x}[n] + B_d u[n] \tag{6.5.16}$$

where the discretized system matrices are

$$A_d = \phi(T) = e^{AT} \tag{6.5.17a}$$

$$B_d = \int_{nT}^{(n+1)T} \phi(nT + T - \tau)\,d\tau\,B \overset{\sigma=nT+T-\tau}{=} -\int_{T}^{0} \phi(\sigma)\,d\sigma\,B = \int_0^T \phi(\tau)\,d\tau\,B \tag{6.5.17b}$$

Here, let us consider another way of computing these system matrices, which is to the taste of digital computers. It comes from making use of the definition of a matrix exponential function in Eq. (6.5.6) to rewrite Eq. (6.5.17) as

$$A_d = e^{AT} = \sum_{m=0}^{\infty} \frac{A^m T^m}{m!} = I + AT\sum_{m=0}^{\infty} \frac{A^m T^m}{(m+1)!} = I + AT\Psi \tag{6.5.18a}$$

$$B_d = \int_0^T \phi(\tau)\,d\tau\,B = \int_0^T \sum_{m=0}^{\infty} \frac{A^m \tau^m}{m!}\,d\tau\,B = \sum_{m=0}^{\infty} \frac{A^m T^{m+1}}{(m+1)!}B = \Psi TB \tag{6.5.18b}$$

where

$$\Psi = \sum_{m=0}^{\infty} \frac{A^m T^m}{(m+1)!}$$
$$\cong I + \frac{AT}{2}\left\{I + \frac{AT}{3}\left\{I + \cdots + \frac{AT}{N-1}\left(I + \frac{AT}{N}\right)\right\}\cdots\right\} \qquad \text{for } N \succ 1 \tag{6.5.19}$$

Now, we apply these discretization formulas for the continuous-time state equation (6.5.3)

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_s(t)$$
$$\text{with } \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ and } u_s(t) = 1 \;\forall\; t \geq 0$$

to get the discretized system matrices and the discretized state equation as

$$\phi(t) = L^{-1}\{[sI - A]^{-1}\} = L^{-1}\left\{\begin{bmatrix} s & -1 \\ 0 & s+1 \end{bmatrix}^{-1}\right\} \overset{(6.5.9)}{=} \begin{bmatrix} 1 & 1 - e^{-t} \\ 0 & e^{-t} \end{bmatrix} \tag{6.5.20a}$$

$$A_d \overset{(6.5.17a)}{=} \phi(T) \overset{(6.5.20a)}{=} \begin{bmatrix} 1 & 1 - e^{-T} \\ 0 & e^{-T} \end{bmatrix} \tag{6.5.20b}$$

$$B_d \overset{(6.5.17b)}{=} \int_0^T \phi(\tau)\, d\tau\, B$$
$$\overset{(6.5.20a)}{=} \int_0^T \begin{bmatrix} 1 & 1 - e^{-\tau} \\ 0 & e^{-\tau} \end{bmatrix} d\tau \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} T - 1 + e^{-T} \\ 1 - e^{-T} \end{bmatrix} \tag{6.5.20c}$$

$$\mathbf{x}[n+1] \overset{(6.5.16)}{=} A_d\mathbf{x}[n] + B_d u[n]$$

$$\begin{bmatrix} x_1[n+1] \\ x_2[n+1] \end{bmatrix} = \begin{bmatrix} 1 & 1 - e^{-T} \\ 0 & e^{-T} \end{bmatrix}\begin{bmatrix} x_1[n] \\ x_2[n] \end{bmatrix} + \begin{bmatrix} T - 1 + e^{-T} \\ 1 - e^{-T} \end{bmatrix} u[n] \tag{6.5.21}$$

We don't need any special algorithm other than an iterative scheme to solve this discrete-time state equation. The formulas (6.5.18a,b) for computing the discretized system matrices are cast into the routine "`c2d_steq()`". The program "`nm652.m`" discretizes the continuous-time state equation (6.5.3) by using the routine and alternatively, the MATLAB built-in routine "`c2d()`". It solves the discretized state equation and plots the results as in Fig. 6.6. As long as the assumption that $u[n] = u(nT)$ for $nT \leq t < (n+1)T$ is valid, the solution $(x[n])$ of the discretized state equation is expected to match that $(x(t))$ of

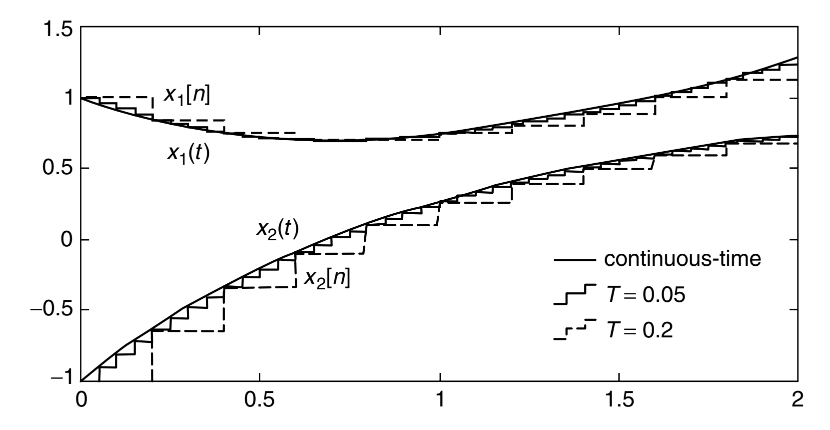


Figure 6.6 The solution of the discretized state equation (6.5.21).

the continuous-time state equation at every sampling instant $t = nT$ and also becomes closer to $x(t)\ \forall\ t$ as the sampling interval T gets shorter (see Fig. 6.6).

```
%nm652.m
% discretize a state eqn x'(t) = Ax(t) + Bu(t) to x[n + 1] = Ad*x[n] + Bd*u[n]
clear, clf
A = [0 1;0 -1]; B = [0;1]; %Eq.(6.5.3)
x0 = [1 -1]; t0 = 0; tf = 2; %initial value and time span
T = 0.2; %sampling interval(period)
eT = exp(-T);
AD = [1 1 - eT; 0 eT]%discretized system matrices obtained analytically
BD = [T + eT - 1; 1 - eT] %Eq.(6.5.21)
[Ad,Bd] = c2d_steq(A,B,T,100) %continuous-to-discrete conversion
[Ad1,Bd1] = c2d(A,B,T) %by the built-in routine
t(1) = 0;  xd(1,:) = x0;  %initial time and initial value
for k = 1:(tf - t0)/T  %solve the discretized state equation
   t(k + 1) = k*T; xd(k + 1,:) = xd(k,:)*Ad' + Bd';
end
stairs([0; t'],[x0; xd]), hold on %stairstep graph
N = 100; t = t0 + [0:N]'*(tf - t0)/N; %time (column) vector
x(:,1) = t-1 + 2*exp(-t); %analytical solution
x(:,2) = 1-2*exp(-t); %Eq.(6.5-12)
plot(t,x)
```

```
function [Ad,Bd] = c2d_steq(A,B,T,N)
if nargin < 4, N = 100; end
I = eye(size(A,2));  PSI = I;
for m = N:-1:1, PSI = I + A*PSI*T/(m + 1); end %Eq.(6.5.19)
Ad = I + A*PSI*T;  Bd = PSI*T*B; %Eq.(6.5.18)
```

6.5.3 High-Order Differential Equation to State Equation

Suppose we are given an Nth-order scalar differential equation together with the initial values of the variable and its derivatives of up to order $N-1$, which is

called an IVP (Initial Value Problem):

$$[\text{IVP}]_N : x^{(N)}(t) = f(t, x(t), x'(t), x^{(2)}(t), \ldots, x^{(N-1)}(t))$$
$$\text{with the initial values } x(t_0) = x_{10}, x'(t_0) = x_{20}, \ldots, x^{(N-1)}(t_0) = x_{N0} \tag{6.5.22}$$

Defining the state vector and the initial state as

$$\mathbf{x}(t) = \begin{bmatrix} x_1 = x \\ x_2 = x' \\ x_3 = x^{(2)} \\ \vdots \\ x_N = x^{(N-1)} \end{bmatrix}, \qquad \mathbf{x}(t_0) = \begin{bmatrix} x_{10} \\ x_{20} \\ x_{30} \\ \vdots \\ x_{N0} \end{bmatrix} \tag{6.5.23}$$

we can rewrite Eq. (6.5.22) in the form of a first-order vector differential equation—that is, a state equation—as

$$\begin{bmatrix} x_1{}'(t) \\ x_2'(t) \\ x_3'(t) \\ \vdots \\ x_N'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ x_3(t) \\ x_4(t) \\ \vdots \\ f(t, x(t), x'(t), x^{(2)}(t), \ldots, x^{(N-1)}(t)) \end{bmatrix}$$
$$\mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x}(t)) \qquad \text{with } \mathbf{x}(t_0) = \mathbf{x}_0 \tag{6.5.24}$$

For example, we can convert a third-order scalar differential equation

$$x^{(3)}(t) + a_2 x^{(2)}(t) + a_1 x'(t) + a_0 x(t) = u(t)$$

into a state equation of the form

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \\ x_3'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -a_0 & -a_1 & -a_2 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u(t) \tag{6.5.25a}$$

$$x(t) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} \tag{6.5.25b}$$

6.5.4 Stiff Equation

Suppose that we are given a vector differential equation involving more than one dependent variable with respect to the independent variable t. If the magnitudes of the derivatives of the dependent variables with respect to t (corresponding

to their changing rates) are significantly different, such a differential equation is said to be stiff because it is difficult to be solved numerically. For such a stiff differential equation, we should be very careful in choosing the step-size in order to avoid numerical instability problem and get a reasonably accurate solution within a reasonable computation time. Why? Because we should use a small step-size to grasp rapidly changing variables, and it requires a lot of computation to cover slowly changing variables for such a long time as it lasts.

Actually, there is no clear distinction between stiff and non-stiff differential equations, since stiffness of a differential equation is a matter of degree. Then, is there any way to estimate the degree of stiffness for a given differential equation? The answer is yes, if the differential equation can be arranged into an LTI state equation like Eq. (6.5.4), the solution of which consists of components having the time constants (modes) equal to the eigenvalues of the system matrix A. For example, the system matrix of Eq. (6.5.3) has the eigenvalues

$$|sI - A| = 0, \quad \det\left\{\begin{bmatrix} s & -1 \\ 0 & s+1 \end{bmatrix}\right\} = s(s+1) = 0, \quad s = 0 \text{ and } s = -1$$

which can be observed as the time constants of two terms $1 = e^{0t}$ and e^{-t} in the solution (6.5.12). In this context, a measure of stiffness is the ratio of the maximum over the minimum among the absolute values of (negative) real parts of the eigenvalues of the system matrix A:

$$\eta(A) = \frac{\text{Max}\{|\text{Re}(\lambda_i)|\}}{\text{Min}\{|\text{Re}(\lambda_i)| \neq 0\}} \tag{6.5.26}$$

This can be thought of as the degree of unbalance between the fast mode and the slow mode.

Now, what we must know is how to handle stiff differential equations. Fortunately, MATLAB has several built-in routines like "`ode15s()`", "`ode23s()`", "`ode23t()`", and "`ode23tb()`", which are fabricated to deal with stiff differential equations efficiently. One may use the `help` command to see their detailed usages. Let's apply them for a Van der Pol equation

$$\frac{d^2y(t)}{dt^2} - \mu(1 - y^2(t))\frac{dy(t)}{dt} + y(t) = 0 \qquad \text{with} \quad y(0) = 2, \frac{dy(t)}{dt} = 0 \tag{6.5.27a}$$

which can be written in the form of a state equation as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ \mu(1 - x_1^2(t))x_2(t) - x_1(t) \end{bmatrix} \quad \text{with} \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \tag{6.5.27b}$$

For this job, we defined this equation in an M-file named "`df_van.m`" and made the MATLAB program "`nm654.m`", where we declared the parameter μ (mu) as

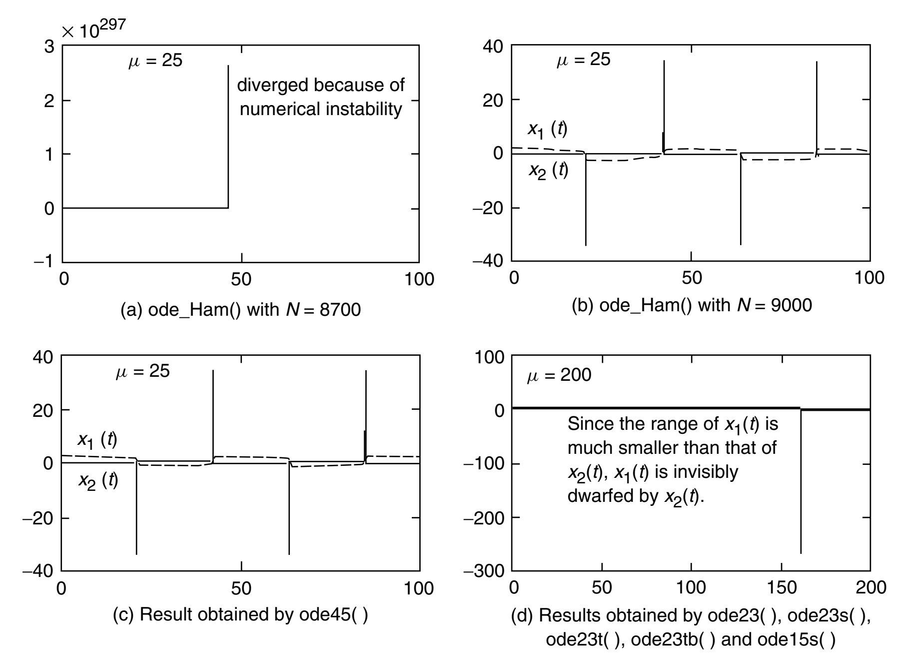


Figure 6.7 Numerical solutions of Van der Pol equation obtained by various routines.

a global variable so that it could be passed on to any related routines/functions as well as "`df_van.m`". In the beginning of the program, we set the global parameter μ to 25 and applied "`ode_Ham()`" with the number of segments $N = 8700$ and 9000. The results are depicted in Figs. 6.7a and 6.7b, which show how crucial the choice of step-size is for a stiff equation. Next, we applied "`ode45()`" to obtain the solution depicted in Fig. 6.7c, which is almost the same as Fig. 6.7b, but with the computation time less than one fourth of that taken by "`ode_Ham()`". This reveals the merit of the MATLAB built-in routines that may save the computation time as well as spare our trouble to choose the step-size, because the step-size is adaptively determined inside the routines. Then, setting $\mu = 200$, we applied the MATLAB built-in routines "`ode45()`"/"`ode23()`"/"`ode15s()`"/"`ode23s()`"/"`ode23t()`"/"`ode23tb()`" to get the results that are little different as depicted in Fig. 6.7d, each taking the computation time as

```
time = 24.9530   14.9690   0.1880   0.2650   0.2500   0.2820
```

The computation time-efficiency of "`ode15s()`"/"`ode23s()`"/"`ode23t()`"/ "`ode23tb()`" (designed deliberately for handling stiff differential equations) over "`ode45()`"/"`ode23()`" becomes prominent as the value of parameter μ (mu) gets large, reflecting high stiffness.

```
%nm654.m
% to solve a stiff differential eqn called Van der Pol equation
global mu
mu=25, t0=0; tf = 100; tspan = [t0 tf]; xo = [2 0];
[tH1,xH1] = ode_Ham('df_van',tspan,x0,8700);
subplot(221), plot(tH1,xH1)
tic,[tH2,xH2] = ode_Ham('df_van',tspan,x0,9000); time_Ham = toc
tic,[t45,x45] = ode45('df_van',tspan,x0); time_o45 = toc
subplot(222), plot(tH2,xH2), subplot(223), plot(t45,x45)
mu = 200; tf = 200; tspan = [t0 tf];
tic,[t45,x45] = ode45('df_van',tspan,x0); time(1) = toc;
tic,[t23,x23] = ode23('df_van',tspan,x0); time(2) = toc;
tic,[t15s,x15s] = ode15s('df_van',tspan,x0); time(3) = toc;
tic,[t23s,x23s] = ode23s('df_van',tspan,x0); time(4) = toc;
tic,[t23t,x23t] = ode23t('df_van',tspan,x0); time(5) = toc;
tic,[t23tb,x23tb] = ode23tb('df_van',tspan,x0); time(6) = toc;
plot(t45,x45, t23,x23, t15s,x15s, t23s,x23s, t23t,x23t, t23tb,x23tb)
disp(' ode23  ode15s  ode23s  ode23t  ode23tb')
time
```

```
function dx = df_van(t,x)
%Van der Pol differential equation (6.5.27)
global mu
dx=zeros(size(x));
dx(1) = x(2); dx(2) = mu*(1-x(1).^2).*x(2) - x(1);
```

6.6 BOUNDARY VALUE PROBLEM (BVP)

A boundary value problem (BVP) is an Nth-order differential equation with some of the values of dependent variable $x(t)$ and its derivative specified at the initial time t_0 and others specified at the final time t_f.

$$[\text{BVP}]_N : \qquad x^{(N)}(t) = f(t, x(t), x'(t), x^{(2)}(t), \ldots, x^{(N-1)}(t))$$
$$\text{with the boundary values } x(t_1) = x_{10}, x'(t_2) = x_{21}, \ldots, x^{(N-1)}(t_N) = x_{N,N-1} \tag{6.6.1}$$

In some cases, some relations between the initial values and the final values may be given as a mixed-boundary condition instead of the initial/final values specified. This section covers the shooting method and the finite difference method that can be used to solve a second-order BVP as

$$[\text{BVP}]_2 : x''(t) = f(t, x(t), x'(t)) \qquad \text{with } x(t_0) = x_0, x(t_f) = x_f \tag{6.6.2}$$

6.6.1 Shooting Method

The idea of this method is to assume the value of $x'(t_0)$, then solve the differential equation (IVP) with the initial condition $[x(t_0)\ x'(t_0)]$ and keep adjusting the value

of $x'(t_0)$ and solving the IVP repetitively until the final value $x(t_f)$ of the solution matches the given boundary value x_f with enough accuracy. It is similar to adjusting the angle of firing a cannon so that the shell will eventually hit the target and that's why this method is named the *shooting* method. This can be viewed as a nonlinear equation problem, if we regard $x'(t_0)$ as an independent variable and the difference between the resulting final value $x(t_f)$ and the desired one x_f as a (mismatching) function of $x'(t_0)$. So the solution scheme can be systemized by using the secant method (Section 4.5) and is cast into the MATLAB routine "`bvp2_shoot()`".

(cf) We might have to adjust the shooting position with the angle fixed, instead of adjusting the shooting angle with the position fixed or deal with the mixed-boundary conditions. See Problems 6.6, 6.7, and 6.8.

For example, let's consider a BVP consisting of the second-order differential equation

$$x''(t) = 2x^2(t) + 4t\ x(t)x'(t) \quad \text{with } x(0) = \frac{1}{4}, x(1) = \frac{1}{3} \tag{6.6.3}$$

```
function [t,x] = bvp2_shoot(f,t0,tf,x0,xf,N,tol,kmax)
%To solve BVP2: [x1,x2]' = f(t,x1,x2) with x1(t0) = x0, x1(tf) = xf
if nargin < 8, kmax = 10; end
if nargin < 7, tol = 1e-8; end
if nargin < 6, N = 100; end
dx0(1) = (xf - x0)/(tf-t0); % the initial guess of x'(t0)
[t,x] = ode_RK4(f,[t0 tf],[x0 dx0(1)],N); % start up with RK4
plot(t,x(:,1)), hold on
e(1) = x(end,1) - xf; % x(tf) - xf: the 1st mismatching (deviation)
dx0(2) = dx0(1) - 0.1*sign(e(1));
for k = 2: kmax-1
   [t,x] = ode_RK4(f,[t0 tf],[x0 dx0(k)],N);
   plot(t,x(:,1))
   %difference between the resulting final value and the target one
   e(k) = x(end,1) - xf; % x(tf)- xf
   ddx = dx0(k) - dx0(k - 1); % difference between successive derivatives
   if abs(e(k))< tol | abs(ddx)< tol, break; end
   deddx = (e(k) - e(k - 1))/ddx; % the gradient of mismatching error
   dx0(k + 1) = dx0(k) - e(k)/deddx; %move by secant method
end
```

```
%do_shoot  to solve BVP2 by the shooting method
t0 = 0; tf = 1;  x0 = 1/4; xf = 1/3; %initial/final times and positions
N = 100; tol = 1e-8; kmax = 10;
[t,x] = bvp2_shoot('df661',t0,tf,x0,xf,N,tol,kmax);
xo = 1./(4 - t.*t); err = norm(x(:,1) - xo)/(N + 1)
plot(t,x(:,1),'b', t,xo,'r') %compare with true solution (6.6.4)
```

```
function dx = df661(t,x) %Eq.(6.6.5)
dx(1) = x(2);  dx(2) = (2*x(1) + 4*t*x(2))*x(1);
```

The solution $x(t)$ and its derivative $x'(t)$ are known as

$$x(t) = \frac{1}{4 - t^2} \quad \text{and} \quad x'(t) = \frac{2t}{(4 - t^2)^2} = 2t\ x^2(t) \tag{6.6.4}$$

Note that this second-order differential equation can be written in the form of state equation as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ 2x_1^2(t) + 4t\ x_1(t)x_2(t) \end{bmatrix} \quad \text{with} \quad \begin{bmatrix} x_1(0) \\ x_2(1) \end{bmatrix} = \begin{bmatrix} x_0 = 1/4 \\ x_f = 1/3 \end{bmatrix} \tag{6.6.5}$$

In order to apply the shooting method, we set the initial guess of $x_2(0) = x'(0)$ to

$$dx0[1] = x_2(0) = \frac{x_f - x_0}{t_f - t_0} \tag{6.6.6}$$

and solve the state equation with the initial condition $[x_1(0)\ \ x_2(0) = dx0[1]]$. Then, depending on the sign of the difference `e(1)` between the final value $x_1(1)$ of the solution and the target final value x_f, we make the next guess $dx0[2]$ larger/smaller than the initial guess $dx0[1]$ and solve the state equation again with the initial condition $[x_1(0)\ \ dx0[2]]$. We can start up the secant method with the two initial values $dx0[1]$ and $dx0[2]$ and repeat the iteration until the difference (error) `e(k)` becomes sufficiently small. For this job, we compose the MATLAB program "`do_shoot.m`", which uses the routine "`bvp2_shoot()`" to get the numerical solution and compares it with the true analytical solution. Figure 6.8 shows that the numerical solution gets closer to the true analytical solution after each round of adjustment.

(Q) Why don't we use the Newton method (Section 4.4)?

(A) Because, in order to use the Newton method in the shooting method, we need IVP solutions instead of function evaluations to find the numerical Jacobian at every iteration, which will require much longer computation time.

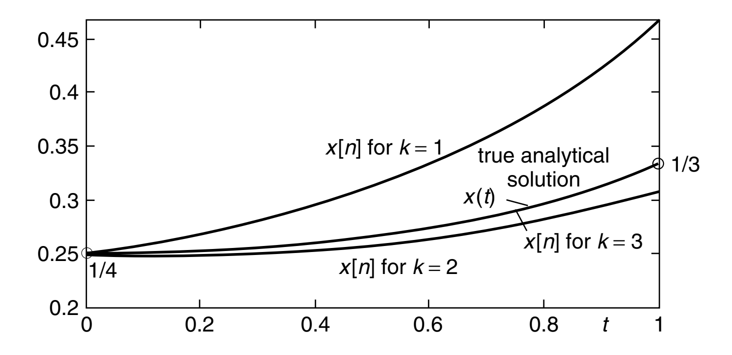


Figure 6.8 The solution of a BVP obtained by using the shooting method.

6.6.2 Finite Difference Method

The idea of this method is to divide the whole interval $[t_0, t_f]$ into N segments of width $h = (t_f - t_0)/N$ and approximate the first & second derivatives in the differential equations for each grid point by the central difference formulas. This leads to a tridiagonal system of equations with respect to $(N - 1)$ variables $\{x_i = x(t_0 + ih), i = 1, \ldots, N - 1\}$. However, in order for this system of equations to be solved easily, it should be linear, implying that its coefficients may not contain any term of x.

For example, let's consider a BVP consisting of the second-order linear differential equation

$$x''(t) + a_1(t)x'(t) + a_0(t)x(t) = u(t) \quad \text{with } x(t_0) = x_0, x(t_f) = x_f \tag{6.6.7}$$

According to the finite difference method, we divide the solution interval $[t_0, t_f]$ into N segments and convert the differential equation for each grid point $t_i = t_0 + ih$ into a difference equation as

$$\frac{x_{i+1} - 2x_i + x_{i-1}}{h^2} + a_{1i}\frac{x_{i+1} - x_{i-1}}{2h} + a_{0i}x_i = u_i$$

$$(2 - ha_{1i})x_{i-1} + (-4 + 2h^2a_{0i})x_i + (2 + ha_{1i})x_{i+1} = 2h^2u_i \tag{6.6.8}$$

Then, taking account of the boundary condition that $x_0 = x(t_0)$ and $x_N = x(t_f)$, we collect all of the $(N - 1)$ equations to construct a tridiagonal system of equations as

$$\begin{bmatrix} -4 + 2h^2a_{01} & 2 + ha_{11} & 0 & \bullet & 0 & 0 & 0 \\ 2 - ha_{12} & -4 + 2h^2a_{02} & 2 + ha_{12} & \bullet & 0 & 0 & 0 \\ 0 & 2 - ha_{13} & -4 + 2h^2a_{03} & \bullet & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & \bullet & -4 + 2h^2a_{0,N-3} & 2 + ha_{1,N-3} & 0 \\ 0 & 0 & 0 & \bullet & 2 - ha_{1,N-2} & -4 + 2h^2a_{0,N-2} & 2 + ha_{1,N-2} \\ 0 & 0 & 0 & \bullet & 0 & 2 - ha_{1,N-1} & -4 + 2h^2a_{0,N-1} \end{bmatrix}$$

$$\times \begin{bmatrix} x_1 \\ x_2 \\ x_2 \\ \bullet \\ x_{N-3} \\ x_{N-2} \\ x_{N-1} \end{bmatrix} = \begin{bmatrix} 2h^2u_1 - (2 - ha_{11})x_0 \\ 2h^2u_2 \\ 2h^2u_3 \\ \bullet \\ 2h^2u_{N-3} \\ 2h^2u_{N-2} \\ 2h^2u_{N-1} - (2 - ha_{1,N-1})x_N \end{bmatrix} \tag{6.6.9}$$

This can be solved efficiently by using the MATLAB routine "`trid()`", which is dedicated to a tridiagonal system of linear equations.

The whole procedure of the finite difference method for solving a second-order linear differential equation with boundary conditions is cast into the MATLAB routine "`bvp2_fdf()`". This routine is designed to accept the two coefficients a_1 and a_0 and the right-hand-side input u of Eq. (6.6.7) as its first three input arguments, where any of those three input arguments can be given as the function name in case the corresponding term is not a numeric value, but a function of time t. We make the program "`do_fdf`" to use this routine for solving the second-order BVP

$$x''(t) + \frac{2}{t}x'(t) - \frac{2}{t^2}x(t) = 0 \quad \text{with } x(1) = 5, x(2) = 3 \tag{6.6.10}$$

```
function [t,x] = bvp2_fdf(a1,a0,u,t0,tf,x0,xf,N)
% solve BVP2: x" + a1*x' + a0*x = u with x(t0) = x0, x(tf) = xf
%  by the finite difference method
h = (tf - t0)/N; h2 = 2*h*h;
t = t0+[0:N]'*h;
if ~isnumeric(a1), a1 = a1(t(2:N)); %if a1 = name of a function of t
 elseif length(a1) == 1, a1 = a1*ones(N - 1,1);
end
if ~isnumeric(a0), a0 = a0(t(2:N)); %if a0 = name of a function of t
 elseif length(a0) == 1, a0 = a0*ones(N - 1,1);
end
if ~isnumeric(u), u = u(t(2:N)); %if u = name of a function of t
 elseif length(u) == 1, u = u*ones(N-1,1);
 else u = u(:);
end
A = zeros(N - 1,N - 1);  b = h2*u;
ha = h*a1(1); A(1,1:2) = [-4 + h2*a0(1)  2 + ha];
b(1) = b(1)+(ha - 2)*x0;
for m = 2:N - 2 %Eq.(6.6.9)
   ha = h*a1(m); A(m,m - 1:m + 1) = [2-ha  -4 + h2*a0(m)  2 + ha];
end
ha = h*a1(N - 1); A(N - 1,N - 2:N - 1) = [2 - ha  -4 + h2*a0(N - 1)];
b(N - 1) = b(N-1)-(ha+2)*xf;
x = [x0 trid(A,b)' xf]';
```

```
function x = trid(A,b)
% solve tridiagonal system of equations
N = size(A,2);
for m = 2:N % Upper Triangularization
  tmp = A(m,m - 1)/A(m - 1,m - 1);
  A(m,m) = A(m,m) -A(m - 1,m)*tmp; A(m,m - 1) = 0;
  b(m,:) = b(m,:) -b(m - 1,:)*tmp;
end
x(N,:) = b(N,:)/A(N,N);
for m = N - 1: -1: 1 % Back Substitution
  x(m,:) = (b(m,:) -A(m,m + 1)*x(m + 1))/A(m,m);
end
```

```
%do_fdf  to solve BVP2 by the finite difference method
clear, clf
t0 = 1; x0 = 5;  tf = 2; xf = 3; N = 100;
a1 = inline('2./t','t'); a0 = inline('-2./t./t','t'); u = 0; %Eq.(6.6.10)
[tt,x] = bvp2_fdf(a1,a0,u,t0,tf,x0,xf,N);
%use the MATLAB built-in command 'bvp4c()'
df = inline('[x(2); 2./t.*(x(1)./t - x(2))]','t','x');
fbc = inline('[x0(1) - 5; xf(1) - 3]','x0','xf');
solinit = bvpinit(linspace(t0,tf,5),[1 10]); %initial solution interval
sol = bvp4c(df,fbc,solinit,bvpset('RelTol',1e-4));
x_bvp = deval(sol,tt); xbv = x_bvp(1,:)';
%use the symbolic computation command 'dsolve()'
xo = dsolve('D2x + 2*(Dx - x/t)/t=0','x(1) = 5, x(2) = 3')
xot = subs(xo,'t',tt); %xot=4./tt./tt +tt; %true analytical solution
err_fd = norm(x - xot)/(N+1) %error between numerical/analytical solution
err_bvp = norm(xbv - xot)/(N + 1)
plot(tt,x,'b',tt,xbv,'r',tt,xot,'k') %compare with analytical solution
```

We run it to get the result depicted in Fig. 6.9 and, additionally, use the symbolic computation command "`dsolve()`" and "`subs()`" to get the analytical solution

$$x(t) = t + \frac{4}{t^2} \tag{6.6.11}$$

and substitute the time vector into the analytical solution to obtain its numeric values for check.

Note the following things about the shooting method and the finite difference method:

- While the shooting method is applicable to linear/nonlinear BVPs, the finite difference method is suitable for linear BVPs. However, we can also apply the finite difference method in an iterative manner to solve nonlinear BVPs (see Problem 6.10).

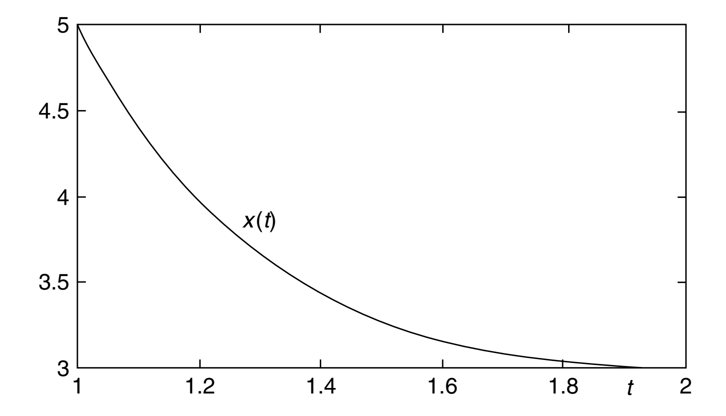


Figure 6.9 A solution of a BVP obtained by using the finite difference method.

- Both methods can be modified to solve BVPs with mixed-boundary conditions (see Problems 6.7 and 6.8).
- In MATLAB 6.x, the "`bvp4c()`" command is available for solving linear/nonlinear BVPs with mixed-boundary conditions (see Problems 6.7–6.10).
- The symbolic computation command "`dsolve()`" introduced in Section 6.5.1 can be used to solve a BVP so long as the differential equation is linear, that is, its coefficients may depend on time t, but not on the (unknown) dependent variable $x(t)$.
- The usages of "`bvp4c()`" and "`dsolve()`" are illustrated in the program "`do_fdf`", where another symbolic computation command "`subs()`" is used to evaluate a symbolic expression at certain value(s) of the variable.

PROBLEMS

6.0 MATLAB Commands `quiver()` and `quiver3()` and Differential Equation

(a) Usage of `quiver()`

Type '`help quiver`' into the MATLAB command window, and then you will see the following program showing you how to use the `quiver()` command for plotting gradient vectors. You can also get Fig. P6.0.1 by running the block of statements in the box below. Try it and note that the size of the gradient vector at each point is proportional to the slope at the point.

```
%do_quiver
[x,y] = meshgrid(-2:.5:2,-1:.25:1);
z = x.*exp(-x.^2 - y.^2);
[px,py]  = gradient(z,.5,.25);
contour(x,y,z), hold on, quiver(x,y,px,py)
axis image %the same as AXIS EQUAL except that
           %the plot box fits tightly around the data
```

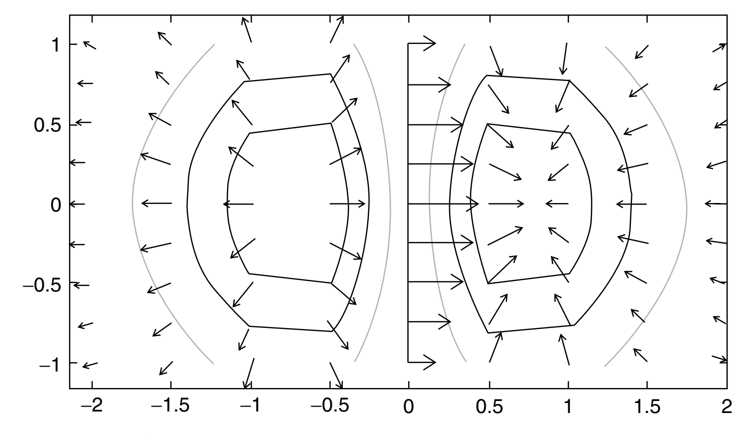


Figure P6.0.1 Graphs obtained by using `gradient()`, `contour()`, `quiver()`.

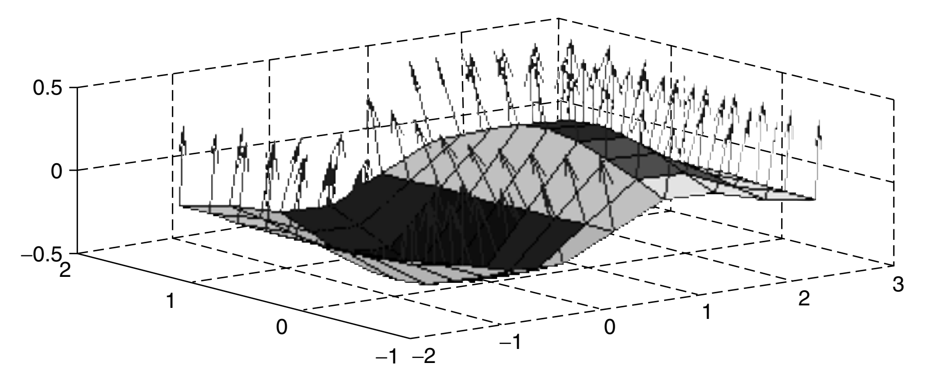


Figure P6.0.2 Graphs obtained by using `surfnorm()`, `quiver3()`, `surf()`.

(b) Usage of `quiver3()`

You can obtain Fig. P6.0.2 by running the block of statements that you see after typing '`help quiver3`' into the MATLAB command window. Note that the "`surfnorm()`" command generates normal vectors at points specified by (x, y, z) on the surface drawn by "`surf()`" and the "`quiver3()`" command plots the normal vectors.

```
%do_quiver3
clear, clf
[x,y] = meshgrid(-2:.5:2,-1:.25:1);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z), hold on
[u,v,w] = surfnorm(x,y,z);
quiver3(x,y,z,u,v,w);
```

(c) Gradient Vectors and One-Variable Differential Equation

We might get the meaning of the solution of a differential equation by using the "`quiver()`" command, which is used in the following program "`do_ode.m`" for drawing the time derivatives at grid points as defined by the differential equation

$$\frac{dy(t)}{dt} = -y(t) + 1 \quad \text{with the initial condition } y(0) = 0 \qquad \text{(P6.0.1)}$$

The slope/direction field together with the numerical solution in Fig. P6.0.3a is obtained by running the program and it can be regarded as a set of possible solution curve segments. Starting from the initial point and moving along the slope vectors, you can get the solution curve. Modify the program and run it to plot the slope/direction

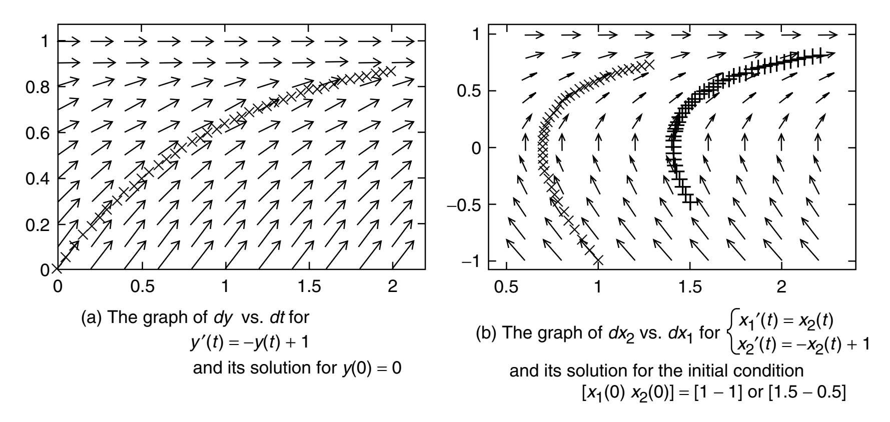


Figure P6.0.3 Possible solutions of differential equation and slope/direction field.

field ($x_2(t)$ versus $x_1(t)$) and the numerical solution for the following differential equation as depicted in Fig. P6.0.3b.

$$\begin{aligned} x_1'(t) &= x_2(t) \\ x_2'(t) &= -x_2(t) + 1 \end{aligned} \quad \text{with} \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{or} \begin{bmatrix} 1.5 \\ -0.5 \end{bmatrix} \tag{P6.0.2}$$

```
%do_ode.m
% This uses quiver() to plot possible solution curve segments
% called the slope/directional field for y'(t) + y = 1
clear, clf
t0 = 0; tf = 2; tspan = [t0 tf]; x0 = 0;
[t,y] = meshgrid(t0:(tf - t0)/10:tf,0:.1:1);
pt = ones(size(t)); py = (1 - y).*pt; %dy = (1 - y)dt
quiver(t,y,pt,py) %y(displacement) vs. t(time)
axis([t0 tf + .2 0 1.05]), hold on
dy=inline('-y + 1','t','y');
[tR,yR] = ode_RK4(dy,tspan,x0,40);
for k = 1:length(tR), plot(tR(k),yR(k),'rx'), pause(0.001); end
```

6.1 A System of Linear Time-Invariant Differential Equations: An LTI State Equation

Consider the following state equation:

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_s(t) \text{ with} \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{P6.1.1}$$

(a) Check the procedure and the result of obtaining the analytical solution by using the Laplace transform technique.

$$X(s) = [sI - A]^{-1}\{\mathbf{x}(0) + BU(s)\}$$
$$= \frac{1}{s(s+3)+2}\begin{bmatrix} s+3 & 1 \\ -2 & s \end{bmatrix}\left\{\begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\frac{1}{s}\right\}$$
$$= \frac{1}{(s+1)(s+2)}\begin{bmatrix} s+3+1/s \\ -2+1 \end{bmatrix}$$
$$= \begin{bmatrix} (s^2+3s+1)/s(s+1)(s+2) \\ -1/(s+1)(s+2) \end{bmatrix}$$

$$X_1(s) = \frac{1/2}{s} + \frac{1}{s+1} - \frac{1/2}{s+2}, \quad x_1(t) = \frac{1}{2} + e^{-t} - \frac{1}{2}e^{-2t} \tag{P6.1.2a}$$

$$X_2(s) = \frac{-1}{s+1} + \frac{1}{s+2}, \quad x_2(t) = -e^{-t} + e^{-2t} \tag{P6.1.2b}$$

(b) Find the numerical solution of the above state equation by using the routine "`ode_RK4()`" (with the number of segments $N = 50$) and the MATLAB built-in routine "`ode45()`". Compare their execution time (by using `tic` and `toc`) and closeness to the analytical solution.

6.2 A Second-Order Linear Time-Invariant Differential Equation

Consider the following second-order differential equation

$$x''(t) + 3x'(t) + 2x(t) = 1 \qquad \text{with } x(0) = 1, x'(0) = 0 \tag{P6.2.1}$$

(a) Check the procedure and the result of obtaining the analytical solution by using the Laplace transform technique.

$$s^2X(s) - x'(0) - sx(0) + 3(sX(s) - x(0)) + 2X(s) = \frac{1}{s}$$

$$X(s) = \frac{s^2+3s+1}{s(s+1)(s+2)}, \qquad x(t) = \frac{1}{2} + e^{-t} - \frac{1}{2}e^{-2t} \tag{P6.2.2}$$

(b) Define the differential equation (P6.2.1) in an M-file so that it can be passed to the MATLAB routines like "`ode_RK4()`" or "`ode45()`" as their input argument (see Section 6.5.1).

6.3 Ordinary Differential Equation and State Equation

(a) Van der Pol Equation
Consider a nonlinear differential equation

$$\frac{d^2}{dt^2}y(t) - \mu(1 - y^2(t))\frac{d}{dt}y(t) + y(t) = 0 \quad \text{with } \mu = 2 \tag{P6.3.1}$$

Compose a program to solve this equation with the initial condition $[y(0)\ y'(0)] = [0.5\ 0]$ and $[-1\ 2]$ for the time interval $[0, 20]$ and plot $y'(t)$ versus $y(t)$ as well as $y(t)$ and $y'(t)$ along the t-axis.

(b) Lorenz Equation: Turbulent Flow and Chaos
Consider a nonlinear state equation.

$$\begin{aligned} x_1'(t) &= \sigma(x_2(t) - x_1(t)) & & \sigma = 10 \\ x_2'(t) &= (1 + \lambda - x_3(t))x_1(t) - x_2(t) & \text{with} \quad & \lambda = 20 \sim 100 \\ x_3'(t) &= x_1(t)x_2(t) - \gamma x_3(t) & & \gamma = 2 \end{aligned} \quad \text{(P6.3.2)}$$

Compose a program to solve this equation with $\lambda = 20$ and 100 for the time interval [0,10] and plot $x_3(t)$ versus $x_1(t)$. Let the initial condition be $[x_1(0)\ x_2(0)\ x_3(0)] = [-8\ -16\ \ 80]$.

(c) Chemical Reactor
Consider a nonlinear state equation describing the concentrations of two reactants and one product in the chemical process.

$$\begin{aligned} x_1'(t) &= a(u_1 - x_1(t)) - bx_1(t)x_2(t) & & a = 5 \\ x_2'(t) &= a(u_2 - x_2(t)) - bx_1(t)x_2(t) & \text{with} \quad & b = 2 \\ x_3'(t) &= -ax_3(t) + bx_1(t)x_2(t) & & u_1 = 3, u_2 = 5 \end{aligned} \quad \text{(P6.3.3)}$$

Compose a program to solve this equation for the time interval [0, 1] and plot $x_1(t)$, $x_2(t)$, and $x_3(t)$. Let the initial condition be $[x_1(0)\ x_2(0)\ x_3(0)] = [1\ 2\ 3]$.

(d) Cantilever Beam: A Differential Equation w.r.t a Spatial Variable
Consider a nonlinear state equation describing the vertical deflection of a beam due to its own weight

$$JE\frac{d^2y}{dx^2} = \rho g\left(1 + \frac{dy}{dx}\right)^2 \left\{x\left(x - \frac{L}{2}\right) + \frac{L^2}{2}\right\} \quad \text{(P6.3.4)}$$

where $JE = 2000\ \text{kg} \cdot \text{m}^3/\text{s}^2$, $\rho = 10$ kg/m, $g = 9.8\ \text{m/s}^2$, $L = 2$ m. Write a program to solve this equation for the interval $[0, L]$ and plot $y(t)$. Let the initial condition be $[y(0)\ \ y'(0)] = [0\ 0]$. Note that the physical meaning of the independent variable for which we usually use the symbol 't' in writing the differential function is not a time, but the x-coordinate of the cantilever beam along the horizontal axis in this problem.

(e) Phase-Locked Loop (PLL)
Consider a nonlinear state equation describing the behavior of a PLL circuit depicted in Fig. P6.3.1.

$$x_1'(t) = \frac{au(t)\cos(x_2(t)) - x_1(t)}{\tau} \quad \text{with} \quad \begin{aligned} a &= 1500 \\ \tau &= 0.002 \\ u(t) &= \sin(\omega_o t) \end{aligned} \quad \text{(P6.3.5a)}$$

$$x_2'(t) = x_1(t) + \omega_c$$

$$y(t) = x_1(t) + \omega_c \quad \text{(P6.3.5b)}$$

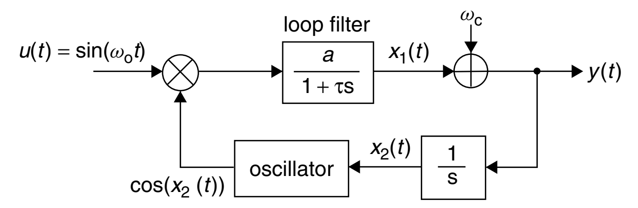


Figure P6.3.1 The block diagram of PLL circuit.

where $\omega_o = 2100\pi$ [rad/s] and $\omega_c = 2000\pi$ [rad/s]. Compose a program to solve this equation for the time interval [0,0.03] and plot $y(t)$ and ω_o. Let the initial condition be $[x_1(0)\ x_2(0)] = [0\ 0]$. Is the output $y(t)$ tracking the frequency ω_o of the input $u(t)$?

(f) DC Motor
Consider a linear differential equation describing the behavior of a DC motor system (Fig. P6.3.2)

$$J\frac{d^2\theta(t)}{dt^2} + B\frac{d\theta(t)}{dt} = T(t) = K_T i(t)$$
$$L\frac{di(t)}{dt} + Ri(t) + K_b\frac{d\theta(t)}{dt} = v(t) \tag{P6.3.6}$$

Convert this system of equations into a first-order vector differential equation—that is, a state equation with respect to the state vector $[\theta(t)\ \theta'(t)\ i(t)]$.

(g) RC Circuit: A Stiff System
Consider a two-mesh RC circuit depicted in Fig. P6.3.3. We can write the mesh equation with respect to the two mesh currents $i_1(t)$ and $i_2(t)$ as

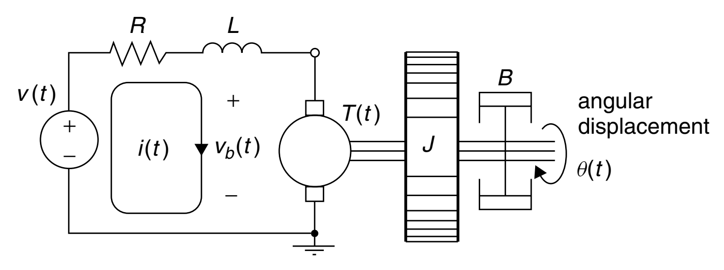


Figure P6.3.2 A DC motor system.

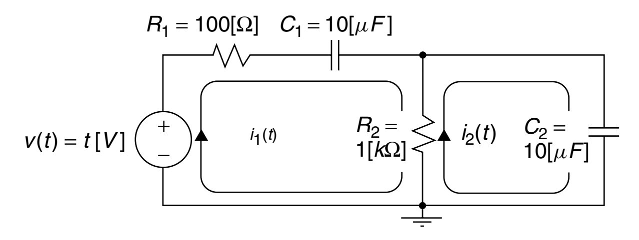


Figure P6.3.3 A two-mesh RC circuit.

$$R_1 i_1(t) + \frac{1}{C_1}\int_{-\infty}^{t} i_1(\tau)\,d\tau + R_2(i_1(t) - i_2(t)) = v(t) = t$$

$$R_2(i_2(t) - i_1(t)) + \frac{1}{C_2}\int_{-\infty}^{t} i_2(\tau)\,d\tau = 0 \qquad \text{(P6.3.7a)}$$

In order to convert this system of differential equations into a state equation, we differentiate both sides and rearrange them to get

$$(R_1 + R_2)\frac{di_1(t)}{dt} - R_2\frac{di_2(t)}{dt} + \frac{1}{C_1}i_1(t) = \frac{dv(t)}{dt} = 1$$

$$-R_2\frac{di_1(t)}{dt} + R_2\frac{di_2(t)}{dt} + \frac{1}{C_2}i_2(t) = 0 \qquad \text{(P6.3.7b)}$$

$$\begin{bmatrix} R_1 + R_2 & -R_2 \\ -R_2 & R_2 \end{bmatrix}\begin{bmatrix} i_1'(t) \\ i_2'(t) \end{bmatrix} = \begin{bmatrix} 1 - i_1(t)/C_1 \\ -i_2(t)/C_2 \end{bmatrix} \qquad \text{(P6.3.7c)}$$

$$\begin{bmatrix} i_1'(t) \\ i_2'(t) \end{bmatrix} = \begin{bmatrix} R_1 + R_2 & -R_2 \\ -R_2 & R_2 \end{bmatrix}^{-1}\begin{bmatrix} 1 - i_1(t)/C_1 \\ -i_2(t)/C_2 \end{bmatrix} \text{ with } G_i = 1/R_i$$

$$= \begin{bmatrix} -G_1/C_1 & -G_1/C_2 \\ -G_1/C_1 & -(G_1 + G_2)/C_2 \end{bmatrix}\begin{bmatrix} i_1(t) \\ i_2(t) \end{bmatrix} + \begin{bmatrix} G_1 \\ G_1 \end{bmatrix}u_s(t) \qquad \text{(P6.3.7d)}$$

where $u_s(t)$ denotes the unit step function whose value is 1 (one) $\forall\ t \geq 0$.

(i) After constructing an M-file function "`df6p03g.m`" which defines Eq. (P6.3.7d) with $R_1 = 100[\Omega]$, $C_1 = 10[\mu F]$, $R_2 = 1[k\Omega]$, $C_2 = 10[\mu F]$, use the MATLAB built-in routines "`ode45()`" and "`ode23s()`" to solve the state equation with the zero initial condition $i_1(0) = i_2(0) = 0$ and plot the numerical solution $i_2(t)$ for $0 \leq t \leq 0.05$ s. For possible change of parameters, you may declare R_1, C_1, R_2, C_2 as global variables both in the function and in the main program named, say, "`nm6p03g.m`". Do you see any symptom of stiffness from the results?

(ii) If we apply the Laplace transform technique to solve this equation with zero initial condition $\mathbf{i}(0) = \mathbf{0}$, we can get

$$\begin{bmatrix} I_1(s) \\ I_2(s) \end{bmatrix} \overset{(6.5.5)}{=} [sI - A]^{-1} Bu(s)$$

$$= \begin{bmatrix} s + G_1/C_1 & G_1/C_2 \\ G_1/C_2 & s + (G_1 + G_2)/C_2 \end{bmatrix}^{-1} \begin{bmatrix} G_1 \\ G_1 \end{bmatrix} \frac{1}{s}$$

$$I_2(s) = \frac{G_1}{s^2 + (G_1/C_1 + (G_1 + G_2)/C_2)s + G_1 G_2/C_1 C_2}$$

$$= \frac{1/100}{s^2 + 2100s + 100000}$$

$$\cong \frac{1/100}{(s + 2051.25)(s + 48.75)}$$

$$\cong \frac{1}{200250}\left(\frac{1}{s + 48.75} - \frac{1}{s + 2051.25}\right)$$

$$i_2(t) \cong \frac{1}{200250}(e^{-48.75t} - e^{-2051.25t}) \tag{P6.3.7e}$$

where $\lambda_1 = -2051.25$ and $\lambda_2 = -48.75$ are actually the eigenvalues of the system matrix A in Eq. (P6.3.7d). Find the measure of stiffness defined by Eq. (6.5.26).

(iii) Using the MATLAB symbolic computation command "`dsolve()`", find the analytical solution of the differential equation (P6.3.7b) and plot $i_2(t)$ together with (P6.3.7e) for $0 \le t \le 0.05$ s. Which of the two numerical solutions obtained in (i) is better? You may refer to the following code:

```
syms R1 R2 C1 C2
i = dsolve('(R1+R2)*Di1 - R2*Di2 + i1/C1 = 1',...
      '-R2*Di1 + R2*Di2 + i2/C2','i1(0) = 0','i2(0) = 0'); %(P6.3.7b)
R1 = 100; R2 = 1000; C1 = 1e-5; C2 = 1e-5;
t0 = 0; tf = 0.05; t = t0+(tf-t0)/100*[0:100];
i2t = eval(i.i2); plot(t,i2t,'m')
```

6.4 Physical Meaning of a Solution for Differential Equation and Its Animation

Suppose we are going to simulate how a vehicle vibrates when it moves with a constant speed on a rugged way, as depicted in Fig. P6.4a. Based on Newton's second law, the situation is modeled by the differential equation (P6.4.1).

$$M\frac{d^2}{dt^2}y(t) + B\frac{d}{dt}(y(t) - u(t)) + K(y(t) - u(t)) = 0 \tag{P6.4.1}$$

$$\text{with } y(0) = 0, \ \ y'(0) = 0$$

```
%do_MBK
clf
t0 = 0; tf = 10; x0 = [0 0];
[t1,x] = ode_Ham('f_MBK',[t0 tf],x0);
dt = t1(2) - t1(1);
for n = 1:length(t1)
  u(n) = udu_MBK(t1(n));
end
figure(1), clf
animation = 1;
if animation
  figure(2), clf
  draw_MBK(5,1,x(1,2),u(1))
  axis([-2 2 -1 14]), axis('equal')
  pause
  for n = 1:length(t1)
    clf, draw_MBK(5,1,x(n,2),u(n),'b')
    axis([-2 2 -1 14]), axis('equal')
    pause(dt)
    figure(1)
    plot(t1(n),u(n),'r.', t1(n),x(n,2),'b.')
    axis([0 tf -0.2 1.2]), hold on
    figure(2)
  end
  draw_MBK(5,1,x(n,2),u(n))
  axis([-2 2 -1 14]), axis('equal')
end
```

```
function [u,du] = udu_MBK(t)
i = fix(t);
if mod(i,2) == 0, u = t-i; du = 1;
 else u = 1 - t + i; du = -1;
end
```

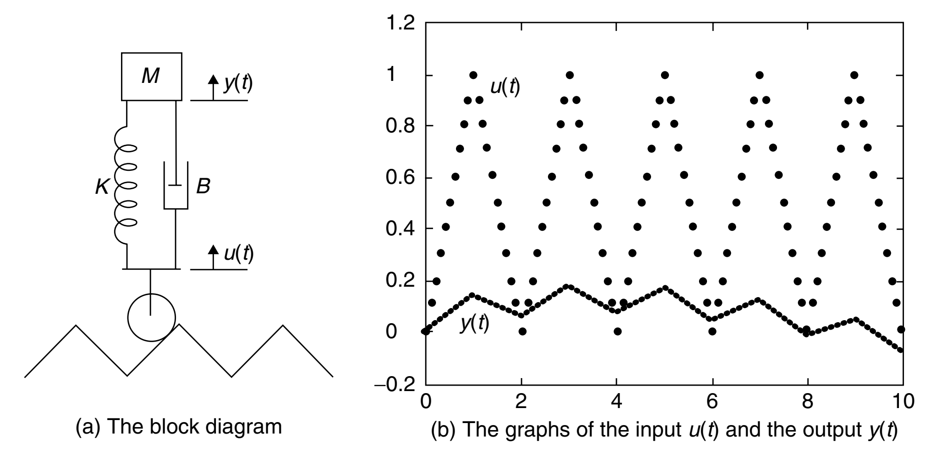
```
function draw_MBK(n,w,y,u,color)
%n: the # of spring windings
%w: the width of each object
%y: displacement of the top of MBK
%u: displacement of the bottom of MBK
if nargin < 5, color = 'k'; end
p1 = [-w  u + 4];  p2 = [-w  9 + y];
xm = 0; ym = (p1(2) + p2(2))/2;
xM = xm + w*1.2*[-1 -1  1  1 -1];
yM = p2(2) + w*[1 3  3  1  1];
plot(xM,yM,color), hold on %Mass
spring(n,p1,p2,w,color) %Spring
damper(xm + w,p1(2),p2(2),w,color) %Damper
wheel_my(xm,p1(2)- 3*w,w,color) %Wheel
```

```
function dx = f_MBK(t,x)
M = 1; B = 0.1; K = 0.1;
[u,du] = udu_MBK(t);
dx = x*[0  1; -B/M - K/M]'+[0 (K*u + B*du)/M];
```

```
function spring(n,p1,p2,w,color)
%draw a spring of n windings, width w from p1 to p2
if nargin < 5, color = 'k'; end
c = (p2(1) - p1(1))/2; d = (p2(2) - p1(2))/2;
f = (p2(1) + p1(1))/2; g = (p2(2) + p1(2))/2;
y = -1:0.01:1;  t = (y+1)*pi*(n + 0.5);
x = -0.5*w*sin(t); y = y+0.15*(1 - cos(t));
a = y(1); b=y(length(x));
y = 2*(y - a)/(b - a)-1;
yyS = d*y - c*x + g;  xxS = x+f;  xxS1 = [f  f];
yyS1 = yyS(length(yyS))+[0 w]; yyS2 = yyS(1)-[0 w];
plot(xxS,yyS,color, xxS1,yyS1,color, xxS1,yyS2,color)
```

```
function damper(xm,y1,y2,w,color)
%draws a damper in (xm-0.5 xm + 0.5 y1 y2)
if nargin < 5, color = 'k'; end
ym = (y1 + y2)/2;
xD1 = xm + w*[0.3*[0  0  -1  1]];  yD1 = [y2 + w  ym  ym  ym];
xD2 = xm + w*[0.5*[-1 -1  1  1]];  yD2 = ym + w*[1  -1  -1
1];
xD3 = xm + [0  0];   yD3 = [y1  ym] - w;
plot(xD1,yD1,color, xD2,yD2,color, xD3,yD3,color)
```

```
function wheel_my(xm,ym,w,color)
%draws a wheel of size w at center (xm,ym)
if nargin < 5, color = 'k'; end
xW1 = xm + w*1.2*[-1  1];  yW1 = ym + w*[2  2];
xW2 = xm*[1  1];   yW2 = ym + w*[2  0];
plot(xW1,yW1,color, xW2,yW2,color)
th = [0:100]/50*pi; plot(xm + j*ym+w*exp(j*th),color)
```



(a) The block diagram

(b) The graphs of the input $u(t)$ and the output $y(t)$

Figure P6.4 A mass–spring–damper system.

where the values of the mass, the viscous friction coefficient, and the spring constant are given as $M = 1$ kg, $B = 0.1$ N s/m, and $K = 0.1$ N/m, respectively. The input to this system is the movement $u(t)$ of the wheel part causing the movement $y(t)$ of the body as the output of the system and is approximated to a triangular wave of height 1 m, duration 1 s, and period 2 s as depicted in Fig. P6.4b. After converting this equation into a state equation as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -K/M & -B/M \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ (B/M)u'(t) + (K/M)u(t) \end{bmatrix} \tag{P6.4.2}$$

$$\text{with } \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

we can use such routines as `ode_Ham()`, `ode45()`,... to solve this state equation and use some graphic functions to draw not only the graphs of $y(t)$ and $u(t)$, but also the animated simulation diagram. You can run the above MATLAB program "`do_MBK.m`" to see the results. Does the suspension system made of a spring and a damper as depicted in Fig. P6.4a absorb effectively the shock caused by the rolling wheel so that the amplitude of vehicle body oscillation is less than 1/5 times that of wheel oscillation?

(cf) If one is interested in graphic visualization with MATLAB, he/she can refer to [N-1].

6.5 A Nonlinear Differential Equation for an Orbit of a Satellite

Consider the problem of an orbit of a satellite, whose position and velocity are obtained as the solution of the following state equation:

$$\begin{aligned} x_1'(t) &= x_3(t) \\ x_2'(t) &= x_4(t) \\ x_3'(t) &= -GM_E x_1(t)/(x_1^2(t) + x_2^2(t))^{3/2} \\ x_4'(t) &= -GM_E x_2(t)/(x_1^2(t) + x_2^2(t))^{3/2} \end{aligned} \tag{P6.5.1}$$

where $G = 6.672 \times 10^{-11}$ N m^2/kg^2 is the gravitational constant, and $M_E = 5.97 \times 10^{24}$ kg is the mass of the earth. Note that (x_1, x_2) and (x_3, x_4) denote the position and velocity, respectively, of the satellite on the plane having the earth at its origin. This state equation is defined in the M-file '`df_sat.m`' below.

(a) Supplement the following program "`nm6p05.m`" which uses the three routines `ode_RK4()`, `ode45()`, and `ode23()` to find the paths of the satellite with the following initial positions/velocities for one day.

```
function dx = df_sat(t,x)
global G Me Re
dx = zeros(size(x));
r = sqrt(sum(x(1:2).^2));
if r <= Re, return; end % when colliding against the earth surface
GMr3 = G*Me/r^3;
dx(1) = x(3); dx(2) = x(4); dx(3) = -GMr3*x(1); dx(4) = -GMr3*x(2);
```

```
%nm6p05.m to solve a nonlinear d.e. on the orbit of a satellite
clear, clf
global G Me Re
G = 6.67e-11; Me = 5.97e24; Re = 64e5;
f = 'df_sat'; ;
t0 = 0; T = 24*60*60; tf = T; N = 2000;
R = 4.223e7;
v20s = [3071 3500 2000];
for iter = 1:length(v20s)
  x10 = R; x20 = 0; v10 = 0; v20 = v20s(iter);
  x0 = [x10 x20 v10 v20]; tol = 1e-6;
  [tR,xR] = ode_RK4(f,[t0 tf],x0,N);
  [t45,x45] = ode45(????????????);
  [t23s,x23s] = ode23s(f,[t0 tf],x0);
  plot(xR(:,1),xR(:,2),'b', x45(:,1),x45(:,2),'k.', ????????????)
  [t45,x45] = ode45(f,[t0 tf],x0,odeset('RelTol',tol));
  [t23s,x23s] = ode23s(????????????????????????????????);
  plot(xR(:,1),xR(:,2),'b', x45(:,1),x45(:,2),'k.', ????????????)
end
```

(i) $(x_{10}, x_{20}) = (4.223 \times 10^7, 0)$[m] and $(x_{30}, x_{40}) = (v_{10}, v_{20}) = (0,\ 3071)$[m/s].

(ii) $(x_{10}, x_{20}) = (4.223 \times 10^7, 0)$[m] and $(x_{30}, x_{40}) = (v_{10}, v_{20}) = (0,\ 3500)$[m/s].

(iii) $(x_{10}, x_{20}) = (4.223 \times 10^7, 0)$[m] and $(x_{30}, x_{40}) = (v_{10}, v_{20}) = (0,\ 2000)$[m/s].

Run the program and check if the plotting results are as depicted in Fig. P6.5.

(b) In Fig. P6.5, we see that the "`ode23s()`" solution path differs from the others for case (ii) and the "`ode45()`" and "`ode23s()`" paths differ from the "`ode_RK4()`" path for case (iii). But, we do not know which one is more accurate. In order to find which one is the closest to the true solution, apply the two routines "`ode45()`" and "`ode23s()`" with smaller relative error tolerance of `tol = 1e-6` to find the paths for the three cases. Which one do you think is the closest to the true solution among the paths obtained in (a)?

(cf) The purpose of this problem is not to compare the several MATLAB routines, but to warn the users of the danger of abusing them. With smaller number of steps (`N`) (i.e., larger step size), the routine "`ode_RK4()`" will also deviate much from the true solution. The MATLAB built-in routines have too many good features to be mentioned here. Note that setting the parameters such as

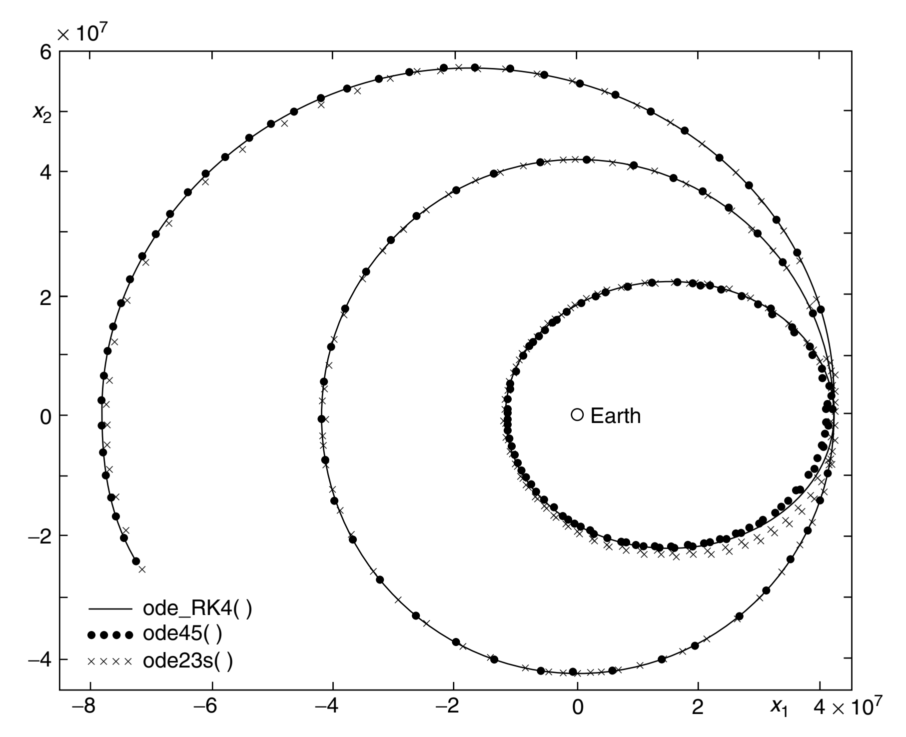


Figure P6.5 The paths of a satellite with the same initial position and different initial velocities.

the relative error tolerance (`RelTol`) is sometimes very important for obtaining a reasonably accurate solution.

6.6 Shooting Method for BVP with Adjustable Position and Fixed Angle

Suppose the boundary condition for a second-order BVP is given as

$$x'(t_0) = x_{20}, \qquad x(t_f) = x_{1f} \tag{P6.6.1}$$

Consider how to modify the MATLAB routines "`bvp2_shoot()`" and "`bvp2_fdf()`" so that they can accommodate this kind of problem.

(a) As for "`bvp2_shootp()`" that you should make, the variable quantity to adjust for improving the approximate solution is not the derivative $x'(t_0)$, but the position $x(t_0)$ and what should be made close to zero is still $f(x(t_0)) = x(t_f) - x_f$. Modify the routine in such a way that $x(t_0)$ is adjusted to make this quantity close to zero and make its declaration part have the initial derivative (`dx0`) instead of the initial position (`x0`) as the fourth input argument as follows.

```
function [t,x] = bvp2_shootp(f,t0,tf,dx0,xf,N,tol,kmax)
```

Noting that the initial derivative of the true solution for Eq. (6.6.3) is zero, apply this routine to solve the BVP by inserting the following statement into the program "do_shoot.m".

```
[t,x1] = bvp2_shootp('df661',t0,tf,0,xf,N,tol,kmax);
```

and plot the result to check if it conforms with that (Fig. 6.8) obtained by "bvp2_shoot()".

(b) As for "bvp2_fdfp()" implementing the finite difference method, you have to approximate the boundary condition as

$$x'(t_0) = x_{20} \rightarrow \frac{x_1 - x_{-1}}{2h} = x_{20}, \qquad x_{-1} = x_1 - 2hx_{20}, \tag{P6.6.2}$$

substitute this into the finite difference equation corresponding to the initial time as

$$\frac{x_1 - 2x_0 + x_{-1}}{h^2} + a_{10}\frac{x_1 - x_{-1}}{2h} + a_{00}x_0 = u_0 \tag{P6.6.3}$$

$$\frac{x_1 - 2x_0 + x_1 - 2hx_{20}}{h^2} + a_{10}x_{20} + a_{00}x_0 = u_0$$

$$(a_{00}h^2 - 2)x_0 + 2x_1 = h^2u_0 + h(2 - ha_{10})x_{20} \tag{P6.6.4}$$

and augment the matrix–vector equation with this equation. Also, make its declaration part have the initial derivative (dx0) instead of the initial position (x0) as the sixth input argument as follows:

```
function [t,x] = bvp2_fdfp(a1,a0,u,t0,tf,dx0,xf,N)
```

Noting that the initial derivative of the true solution for Eq. (6.6.10) is −7, apply this routine to solve the BVP by inserting the following statement into the program "do_fdf.m".

```
[t,x1] = bvp2_fdfp(a1,a0,u,t0,tf,-7,xf,N);
```

and plot the result to check if it conforms with that obtained by using "bvp2_fdf()" and depicted in Fig. 6.9.

6.7 BVP with Mixed-Boundary Conditions I

Suppose the boundary condition for a second-order BVP is given as

$$x(t_0) = x_{10}, \qquad c_1x(t_f) + c_2x'(t_f) = c_3 \tag{P6.7.1}$$

Consider how to modify the MATLAB routines "bvp2_shoot()" and "bvp2_fdf()" so that they can accommodate this kind of problem.

(a) As for "bvp2_shoot()" that you should modify, the variable quantity to adjust for improving the approximate solution is still the derivative $x'(t_0)$, but what should be made close to zero is

$$f(x'(t_0)) = c_1 x(t_f) + c_2 x'(t_f) - c_3 \qquad \text{(P6.7.2)}$$

If you don't know where to begin, modify the routine "bvp2_shoot()" in such a way that $x'(t_0)$ is adjusted to make this quantity close to zero. Regarding the quantity (P6.7.2) as a function of $x'(t_0)$, you may feel as if you were going to solve a nonlinear equation $f(x'(t_0)) = 0$. Here are a few hints for this job:

- Make the declaration part have the boundary coefficient vector cf = [c1 c2 c3] instead of the final position (xf) as the fifth input argument as follows.

```
function [t,x] = bvp2m_shoot(f,t0,tf,x0,cf,N,tol,kmax)
```

- Pick up the first two guesses of $x'(t_0)$ arbitrarily.
- You may need to replace a couple of statements in "bvp2_shoot()" by

```
e(1) = cf*[x(end,:)';-1];
e(k) = cf*[x(end,:)';-1];
```

Now that you have the routine "bvp2m_shoot()" of your own making, don't hesitate to try using the weapon to attack the following problem:

$$x''(t) - 4t\, x(t)x'(t) + 2x^2(t) = 0 \quad \text{with } x(0) = \frac{1}{4},\ 2x(1) - 3x'(1) = 0 \qquad \text{(P6.7.3)}$$

For this job, you only have to modify one statement of the program "do_shoot" (Section 6.6.1) into

```
[t,x] = bvp2m_shoot('df661',t0,tf,x0,[2 -3 0],N,tol,kmax);
```

If you run it to obtain the same solution as depicted in Fig. 6.8, you deserve to be proud of yourself having this book as well as MATLAB; otherwise, just keep trying until you succeed.

(b) As for "bvp2_fdf()" that you should modify, you have only to augment the matrix–vector equation with one row corresponding to the approximate version of the boundary condition $c_1 x(t_f) + c_2 x'(t_f) = c_3$, that is,

$$c_1 x_N + c_2 \frac{x_N - x_{N-1}}{h} = c_3;\ -c_2 x_{N-1} + (c_1 h + c_2) x_N = c_3 h \qquad \text{(P6.7.4)}$$

Needless to say, you should increase the dimension of the matrix `A` to N and move the x_N term on the right-hand side of the $(N-1)$th row back to the left-hand side by incorporating the corresponding statement into the `for` loop. What you have to do with "`bvp2m_fdf()`" for this job is as follows:

- Make the declaration part have the boundary coefficient vector `cf = [c1 c2 c3]` instead of the final position (`xf`) as the seventh input argument.

```
function [t,x] = bvp2m_fdf(a1,a0,u,t0,tf,x0,cf,N)
```

- Replace some statement by `A = zeros(N,N)`.
- Increase the last index of the `for` loop to `N-1`.
- Replace the statements corresponding to the $(N-1)$th row equation by

```
A(N,N-1:N) = [-cf(2) cf(1)*h + cf(2)];    b(N) = cf(3)*h;
```

  which implements Eq. (P6.7.4).
- Modify the last statement arranging the solution as

```
x = [x0 trid(A,b)']';
```

  Now that you have the routine "`bvp2m_fdf()`" of your own making, don't hesitate to try it on the following problem:

$$x''(t) + \frac{2}{t}x'(t) - \frac{2}{t^2}x(t) = 0 \qquad \text{with } x(1) = 5, x(2) + x'(2) = 3 \tag{P6.7.5}$$

  For this job, you only have to modify one statement of the program "`do_fdf.m`" (Section 6.6.2) into

```
[t,x] = bvp2m_fdf(a1,a0,u,t0,tf,x0,[1 1 3],N);
```

  You might need to increase the number of segments `N` to improve the accuracy of the numerical solution. If you run it to obtain the same solution as depicted in Fig. 6.9, be happy with it.

6.8 BVP with Mixed-Boundary Conditions II

Suppose the boundary condition for a second-order BVP is given as

$$c_{01}x(t_0) + c_{02}x'(t_0) = c_{03} \tag{P6.8.1a}$$

$$c_{f1}x(t_f) + c_{f2}x'(t_f) = c_{f3} \tag{P6.8.1b}$$

Consider how to modify the MATLAB routines "`bvp2m_shoot()`" and "`bvp2m_fdf()`" so that they can accommodate this kind of problems.

(a) As for "bvp2mm_shoot()" that you should make, the variable quantity to be adjusted for improving the approximate solution is $x'(t_0)$ or $x(t_0)$ depending on whether or not $c_{01} \neq 0$, while the quantity to be made close to zero is still

$$f(x(t_0), x'(t_0)) = c_{f1}x(t_f) + c_{f2}x'(t_f) - c_{f3} \tag{P6.8.2}$$

If you don't have your own idea, modify the routine "bvp2m_shoot()" in such a way that $x'(t_0)$ or $x(t_0)$ is adjusted to make this quantity close to zero and $x(t_0)$ or $x'(t_0)$ is set by (P6.8.1a), making its declaration as

```
function [t,x] = bvp2mm_shoot(f,t0,tf,c0,cf,N,tol,kmax)
```

where the boundary coefficient vectors `c0 = [c01 c02 c03]` and `cf = [cf1 cf2 cf3]` are supposed to be given as the fourth and fifth input arguments, respectively.

Now that you get the routine "bvp2mm_shoot()" of your own making, try it on the following problem:

$$x''(t) - \frac{2t}{t^2+1}x'(t) + \frac{2}{t^2+1}x(t) = t^2 + 1 \tag{P6.8.3}$$
$$\text{with } x(0) + 6x'(0) = 0, \;\; x(1) + x'(1) = 0$$

(b) As for "bvp2_fdf()" implementing the finite difference method, you only have to augment the matrix–vector equation with two rows corresponding to the approximate versions of the boundary conditions $c_{01}x(t_0) + c_{02}x'(t_0) = c_{03}$ and $c_{f1}x(t_f) + c_{f2}x'(t_f) = c_{f3}$, that is,

$$c_{01}x_0 + c_{02}\frac{x_1 - x_0}{h} = c_{03}, \quad (c_{01}h - c_{02})x_0 + c_{02}x_1 = c_{03}h \tag{P6.8.4a}$$

$$c_{f1}x_N + c_{f2}\frac{x_N - x_{N-1}}{h} = c_{f3}; \quad -c_{f2}x_{N-1} + (c_{f1}h + c_{f2})x_N = c_{f3}h \tag{P6.8.4b}$$

Now that you have the routine "bvp2mm_fdf()" of your own making, try it on the problem described by Eq. (P6.8.3).

(c) Overall, you will need to make the main programs like "nm6p08a.m" and "nm6p08b.m" that apply the routines "bvp2mm_shoot()" and "bvp2mm_fdf()" to get the numerical solutions of Eq. (P6.8.3) and plot them. Additionally, use the MATLAB routine "bvp4c()" to get another solution and plot it together for cross-check.

6.9 Shooting Method and Finite Difference Method for Linear BVPs

Apply the routines "bvp2_shoot()", "bvp2_fdf()", and "bvp4c()" to solve the following BVPs.

```
%nm6p08a.m: to solve BVP2 with mixed boundary conditions
%x" = (2t/t^2 + 1)*x' -2/(t^2+1)*x +t^2+1
%  with x(0)+6x'(0) = 0, x'(1) + x(1) = 0
%shooting method
f = inline('[x(2); 2*(t*x(2) - x(1))./(t.^2 + 1)+(t.^2 + 1)]','t','x');
t0 = 0; tf = 1; N = 100; tol = 1e-8; kmax = 10;
c0 = [1 6 0]; cf = [1 1 0]; %coefficient vectors of boundary condition
[tt,x_sh] = bvp2mm_shoot(f,t0,tf,c0,cf,N,tol,kmax);
plot(tt,x_sh(:,1),'b')
```

```
%nm6p08b.m: finite difference method
a1 = inline('-2*t./(t.^2+1)','t');  a0 = inline('2./(t.^2+1)','t');
u = inline('t.^2+1','t');
t0 = 0; tf = 1; N = 500;
c0 = [1 6 0]; cf = [1 1 0]; %coefficient vectors of boundary condition
[tt,x_fd] = bvp2mm_fdf(a1,a0,u,t0,tf,c0,cf,N);
plot(tt,x_fd,'r')
```

$$y''(x) = f(y'(x), y(x), u(x)) \quad \text{with } y(x_0) = y_0, y(x_f) = y_f \qquad \text{(P6.9.0a)}$$

Plot the solutions and fill in Table P6.9 with the mismatching errors (of the numerical solutions) that are defined as

```
function err = err_of_sol_de(df,t,x,varargin)
% evaluate the error of solutions of differential equation
[Nt,Nx] = size(x); if Nt < Nx, x = x.'; [Nt,Nx] = size(x); end
n1 = 2:Nt - 1; t=t(:); h2s = t(n1 + 1)-t(n1-1);
dx = (x(n1 + 1,:) - x(n1 - 1,:))./(h2s*ones(1,Nx));
num = x(n1 + 1,:)-2*x(n1,:) + x(n1 - 1,:); den = (h2s/2).^2*ones(1,Nx);
d2x = num./den;
for m = 1:Nx
  for n = n1(1):n1(end)
    dfx = feval(df,t(n),[x(n,m) dx(n - 1,m)],varargin{:});
    errm(n - 1,m) = d2x(n - 1,m) - dfx(end);
  end
end
err=sum(errm.^2)/(Nt - 2);
```

```
%nm6p09_1.m
%y"-y'+y = 3*e^2t-2sin(t) with y(0) = 5 & y(2)=-10
t0 = 0; tf = 2; y0 = 5; yf = -10; N = 100; tol = 1e-6; kmax = 10;
df = inline('[y(2); y(2) - y(1)+3*exp(2*t)-2*sin(t)]','t','y');
a1 = -1; a0 = 1; u = inline('3*exp(2*t) - 2*sin(t)','t');
solinit = bvpinit(linspace(t0,tf,5),[-10 5]); %[1 9]
fbc = inline('[y0(1) - 5; yf(1) + 10]','y0','yf');
% Shooting method
tic, [tt,y_sh] = bvp2_shoot(df,t0,tf,y0,yf,N,tol,kmax); times(1) = toc;
% Finite difference method
tic, [tt,y_fd] = bvp2_fdf(a1,a0,u,t0,tf,y0,yf,N); times(2) = toc;
% MATLAB built-in function bvp4c
sol = bvp4c(df,fbc,solinit,bvpset('RelTol',1e-6));
tic, y_bvp = deval(sol,tt); times(3) = toc
% Eror evaluation
ys=[y_sh(:,1) y_fd y_bvp(1,:)']; plot(tt,ys)
err=err_of_sol_de(df,tt,ys)
```

Table P6.9 Comparison of the BVP Solver Routines bvp2_shoot()/bvp2_fdf()

BVP	Routine	Mismatching Error (P6.9.0b)	Times
(P6.9.1) N = 100, tol = 1e-6, kmax = 10	bvp2_shoot()	1.5×10^{-6}	
	bvp2_fdf()		
	bvp4c()	2.9×10^{-6}	
(P6.9.2) N = 100, tol = 1e-6, kmax = 10	bvp2_shoot()		
	bvp2_fdf()	1.6×10^{-23}	
	bvp4c()		
(P6.9.3) N = 100, tol = 1e-6, kmax = 10	bvp2_shoot()	1.7×10^{-17}	
	bvp2_fdf()		
	bvp4c()	7.8×10^{-14}	
(P6.9.4) N = 100, tol = 1e-6, kmax = 10	bvp2_shoot()		
	bvp2_fdf()	4.4×10^{-27}	
	bvp4c()		
(P6.9.5) N = 100, tol = 1e-6, kmax = 10	bvp2_shoot()	8.9×10^{-9}	
	bvp2_fdf()		
	bvp4c()	8.9×10^{-7}	
(P6.9.6) N = 100, tol =1e-6, kmax =10	bvp2_shoot()		
	bvp2_fdf()	4.4×10^{-25}	
	bvp4c()		

$$\text{err} = \frac{1}{N-1}\sum_{i=1}^{N-1}\{D^{(2)}y(x_i) - f(Dy(x_i), y(x_i), u(x_i))\}^2 \tag{P6.9.0b}$$

with

$$D^{(2)}y(x_i) = \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1})}{h^2}, \quad Dy(x_i) = \frac{y(x_{i+1}) - y(x_{i-1})}{2h} \tag{P6.9.0c}$$

$$x_i = x_0 + ih, \qquad h = \frac{x_f - x_0}{N} \tag{P6.9.0d}$$

and can be computed by using the following routine "err_of_sol_de()".

Overall, which routine works the best for linear BVPs among the three routines?

(a) $y''(x) = y'(x) - y(x) + 3e^{2x} - 2\sin x$ with $y(0) = 5,\ y(2) = -10$ (P6.9.1)

(b) $y''(x) = -4y(x)$ with $y(0) = 5,\ y(1) = -5$ (P6.9.2)

(c) $y''(t) = 10^{-6}y(t) + 10^{-7}(t^2 - 50t)$ with $y(0) = 0,\ y(50) = 0$ (P6.9.3)

(d) $y''(t) = -2y(t) + \sin t$ with $y(0) = 0,\ y(1) = 0$ (P6.9.4)

(e) $y''(x) = y'(x) + y(x) + e^x(1 - 2x)$ with $y(0) = 1,\ y(1) = 3e$ (P6.9.5)

(f) $\dfrac{d^2y(r)}{dr^2} + \dfrac{1}{r}\dfrac{dy(r)}{dr} = 0$ with $y(1) = \ln 1,\ y(2) = \ln 2$ (P6.9.6)

6.10 Shooting Method and Finite Difference Method for Nonlinear BVPs

(a) Consider a nonlinear boundary value problem of solving

$$\frac{d^2T}{dx^2} = 1.9 \times 10^{-9}(T^4 - T_a^4),\ \ T_a = 400 \tag{P6.10.1}$$

with the boundary condition $T(x_0) = T_0,\ \ T(x_f) = T_f$

to find the temperature distribution $T(x)$ [°K] in a rod 4 m long, where $[x_0, x_f] = [0, 4]$.

Apply the routines "`bvp2_shoot()`", "`bvp2_fdf()`", and "`bvp4c()`" to solve this differential equation for the two sets of boundary conditions $\{T(0) = 500,\ T(4) = 300\}$ and $\{T(0) = 550,\ T(4) = 300\}$ as listed in Table P6.10. Fill in the table with the mismatching errors defined by Eq. (P6.9.0b) for the three numerical solutions

```
%nm6p10a
clear, clf
K = 1.9e-9; Ta = 400; Ta4 = Ta^4;
df = inline('[T(2); 1.9e-9*(T(1).^4-256e8)]','t','T');
x0 = 0; xf = 4; T0 = 500; Tf = 300; N = 500; tol = 1e-5; kmax = 10;
% Shooting method
[xx,T_sh] = bvp2_shoot(df,x0,xf,T0,Tf,N,tol,kmax);
% Iterative finite difference method
a1 = 0; a0 = 0;  u = T0 + [1:N - 1]*(Tf - T0)/N;
for i = 1:100
  [xx,T_fd] = bvp2_fdf(a1,a0,u,x0,xf,T0,Tf,N);
  u = K*(T_fd(2:N).^4 - Ta4); %RHS of (P6.10.1)
  if i > 1 & norm(T_fd - T_fd0)/norm(T_fd0) < tol, i, break; end
  T_fd0 = T_fd;
end
% MATLAB built-in function bvp4c
solinit = bvpinit(linspace(x0,xf,5),[Tf T0]);
fbc = inline('[Ta(1)-500; Tb(1)-300]','Ta','Tb');
tic, sol = bvp4c(df,fbc,solinit,bvpset('RelTol',1e-6));
T_bvp = deval(sol,xx); time_bvp = toc;
% The set of three solutions
Ts = [T_sh(:,1) T_fd T_bvp(1,:)'];
% Evaluates the errors and plot the graphs of the solutions
err = err_of_sol_de(df,xx,ys)
subplot(321), plot(xx,Ts)
```

Table P6.10 Comparison of the BVP routines `bvp2_shoot()`/`bvp2_fdf()`

Boundary Condition	Routine	Mismatching Error (P6.9.0b)	Time (seconds)
(P6.10.1) with $T_a = 400$ $T(0) = 500,\ T(4) = 300$	`bvp2_shoot()`		
	`bvp2_fdf()`	3.6×10^{-6}	
	`bvp4c()`		
(P6.10.1) with $T_a = 400$ $T(0) = 550,\ T(4) = 300$	`bvp2_shoot()`	NaN (divergent)	N/A
	`bvp2_fdf()`		
	`bvp4c()`	30×10^{-5}	
(P6.10.2) with $y(0) = 0,\ y(1) = 0$	`bvp2_shoot()`		
	`bvp2_fdf()`	3.2×10^{-13}	
	`bvp4c()`		
(P6.10.3) with $y(1) = 4,\ y(2) = 8$	`bvp2_shoot()`	NaN (divergent)	N/A
	`bvp2_fdf()`		
	`bvp4c()`	3.5×10^{-6}	
(P6.10.4) with $y(1) = 1/3,\ y(4) = 20/3$	`bvp2_shoot()`		
	`bvp4c()`	3.4×10^{-10}	
	`bvp2_fdf(c)`		
(P6.10.5) with $y(0) = \pi/2,\ y(2) = \pi/4$	`bvp2_shoot()`	3.7×10^{-14}	
	`bvp2_fdf()`		
	`bvp4c()`	2.2×10^{-9}	
(P6.10-6) with $y(2) = 2, y'(8) = 1/4$	`bvp2_shoot()`		
	`bvp2_fdf()`	5.0×10^{-14}	
	`bvp4c()`		

$$\{T(x_i),\ i = 0 : N\}\ (x_i = x_0 + ih = x_0 + i\frac{x_f - x_0}{N} \text{ with } N = 500$$

Note that the routine “`bvp2_fdf()`” should be applied in an iterative way to solve a nonlinear BVP, because it has been fabricated to accommodate only linear BVPs. You may start with the following program “`nm6p10a.m`”. Which routine works the best for the first case and the second case, respectively?

(b) Apply the routines "`bvp2_shoot()`", "`bvp2_fdf()`", and "`bvp4c()`" to solve the following BVPs. Fill in Table P6.10 with the mismatching errors defined by Eq. (P6.9.0b) for the three numerical solutions and plot the solution graphs if they are reasonable solutions.

(i) $y'' - e^y = 0 \quad \text{with } y(0) = 0, y(1) = 0$ (P6.10.2)

(ii) $y'' - \frac{1}{t}y' - \frac{2}{y}(y')^2 = 0 \quad \text{with } y(1) = 4, y(2) = 8$ (P6.10.3)

(iii) $y'' - \frac{2}{y'+1} = 0 \quad \text{with } y(1) = \frac{1}{3}, y(4) = \frac{20}{3}$ (P6.10.4)

(iv) $y'' = t(y')^2 \quad \text{with } y(0) = \pi/2, y(2) = \pi/4$ (P6.10.5)

(v) $y'' + \frac{1}{y^2}y' = 0 \quad \text{with } y(2) = 2, y'(8) = 1/4$ (P6.10.6)

Especially for the BVP (P6.10.6), the routine "`bvp2m_shoot()`" or "`bvp2mm_shoot()`" developed in Problems 6.7 and 6.8 should be used instead of "`bvp2_shoot()`", since it has a mixed-boundary condition I.

(cf) Originally, the shooting method was developed for solving nonlinear BVPs, while the finite difference method is designed as a one-shot method for solving linear BVPs. But the finite difference method can also be applied in an iterative way to handle nonlinear BVPs, producing more accurate solutions in less computation time.

6.11 Eigenvalue BVPs

(a) A Homogeneous Second-Order BVP to an Eigenvalue Problem
Consider an eigenvalue boundary value problem of solving

$$y''(x) + \omega^2 y = 0 \tag{P6.11.1}$$

$$\text{with } c_{01}y(x_0) + c_{02}y'(x_0) = 0, \;\; c_{f1}y(x_f) + c_{f2}y'(x_f) = 0$$

to find $y(x)$ for $x \in [x_0, x_f]$ with the (possible) angular frequency ω.

In order to use the finite difference method, we divide the solution interval $[x_0, x_f]$ into N subintervals to have the grid points $x_i = x_0 + ih = x_0 + i(x_f - x_0)/N$ and then, replace the derivatives in the differential equation and the boundary conditions by their finite difference approximations (5.3.1) and (5.1.8) to write

$$\frac{y_{i-1} - 2y_i + y_{i+1}}{h^2} + \omega^2 y_i = 0$$

$$y_{i-1} - (2 - \lambda)y_i + y_{i+1} = 0 \qquad \text{with} \quad \lambda = h^2\,\omega^2 \tag{P6.11.2}$$

with

$$c_{01}y_0 + c_{02}\frac{y_1 - y_{-1}}{2h} = 0 \rightarrow y_{-1} = 2h\frac{c_{01}}{c_{02}}y_0 + y_1 \quad \text{(P6.11.3a)}$$

$$c_{f1}y_N + c_{f2}\frac{y_{N+1} - y_{N-1}}{2h} = 0 \rightarrow y_{N+1} = y_{N-1} - 2h\frac{c_{f1}}{c_{f2}}y_N \quad \text{(P6.11.3b)}$$

Substituting the discretized boundary condition (P6.11.3) into (P6.11.2) yields

$$y_{-1} - 2y_0 + y_1 = -\lambda y_0 \xrightarrow{\text{(P6.11.3a)}}$$

$$\left(2 - 2h\frac{c_{01}}{c_{02}}\right)y_0 - 2y_1 = \lambda y_0 \quad \text{(P6.11.4a)}$$

$$y_{i-1} - 2y_i + y_{i+1} = -\lambda y_i \rightarrow -y_{i-1} + 2y_i - y_{i+1} = \lambda y_i$$

$$\text{for} \quad i = 1 : N-1 \quad \text{(P6.11.4b)}$$

$$y_{N-1} - 2y_N + y_{N+1} = -\lambda y_N \xrightarrow{\text{(P6.11.3b)}}$$

$$-2y_{N-1} + \left(2 + 2h\frac{c_{f1}}{c_{f2}}\right)y_N = \lambda y_N \quad \text{(P6.11.4c)}$$

which can be formulated in a compact form as

$$\begin{bmatrix} 2 - 2hc_{01}/c_{02} & -2 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -2 & 2 + 2hc_{f1}/c_{f2} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \cdot \\ y_{N-1} \\ y_N \end{bmatrix}$$

$$= \lambda \begin{bmatrix} y_0 \\ y_1 \\ \cdot \\ y_{N-1} \\ y_N \end{bmatrix}$$

$$A\mathbf{y} = \lambda\mathbf{y}; \qquad [A - \lambda I]\mathbf{y} = \mathbf{0} \quad \text{(P6.11.5)}$$

For this equation to have a nontrivial solution $\mathbf{y} \neq \mathbf{0}$, λ must be one of the eigenvalues of the matrix A and the corresponding eigenvectors are possible solutions.

```
function [x,Y,ws,eigvals] = bvp2_eig(x0,xf,c0,cf,N)
% use the finite difference method to solve an eigenvalue BVP4:
%  y"+w^2*y = 0 with c01y(x0) + c02y'(x0) = 0, cf1y(xf) + cf2y'(xf) = 0
%input: x0/xf = the initial/final boundaries
%       c0/cf = the initial/final boundary condition coefficients
%       N - 1 = the number of internal grid points.
%output: x    = the vector of grid points
%        Y    = the matrix composed of the eigenvector solutions
%        ws   = angular frequencies corresponding to eigenvalues
%     eigvals = the eigenvalues
if nargin < 5|N < 3, N = 3; end
h = (xf - x0)/N; h2 = h*h; x = x0+[0:N]*h;
N1 = N + 1;
if abs(c0(2)) < eps, N1 = N1 - 1; A(1,1:2) = [2 -1];
 else A(1,1:2) = [2*(1-c0(1)/c0(2)*h) -2];           %(P6.11.4a)
end
if abs(cf(2)) < eps, N1 = N1 - 1; A(N1,N1 - 1:N1) = [-1 2];
 else A(N1,N1 - 1:N1) = [-2 2*(1 + cf(1)/cf(2)*h)];   %(P6.11.4c)
end
if N1 > 2
  for m = 2:ceil(N1/2), A(m,m - 1:m + 1) = [-1 2 -1]; end  %(P6.11.4b)
end
for m=ceil(N1/2) + 1:N1 - 1, A(m,:) = fliplr(A(N1 + 1 - m,:)); end
[V,LAMBDA] = eig(A); eigvals = diag(LAMBDA)';
[eigvals,I] = sort(eigvals); % sorting in the ascending order
V = V(:,I);
ws = sqrt(eigvals)/h;
if abs(c0(2)) < eps, Y = zeros(1,N1); else Y = []; end
Y = [Y; V];
if abs(cf(2)) < eps, Y = [Y; zeros(1,N1)]; end
```

Note the following things:

- The angular frequency corresponding to the eigenvalue λ can be obtained as

$$\omega = \sqrt{\lambda/a_0}/h \tag{P6.11.6}$$

- The eigenvalues and the eigenvectors of a matrix A can be obtained by using the MATLAB command '`[V,D] = eig(A)`'.
- The above routine "`bvp2_eig()`" implements the above-mentioned scheme to solve the second-order eigenvalue problem (P6.11.1).
- In particular, a second-order eigenvalue BVP

$$y''(x) + \omega^2 y = 0 \qquad \text{with } y(x_0) = 0,\, y(x_f) = 0 \tag{P6.11.7}$$

  corresponds to (P6.11.1) with $\mathbf{c}_0 = [c_{01}\ \ c_{02}] = [1 \quad 0]$ and $\mathbf{c}_f = [c_{f1}\ c_{f2}] = [1\ 0]$ and has the following analytical solutions:

$$y(x) = a\sin\omega x \quad \text{with } \omega = \frac{k\pi}{x_f - x_0},\, k = 1, 2, \ldots \tag{P6.11.8}$$

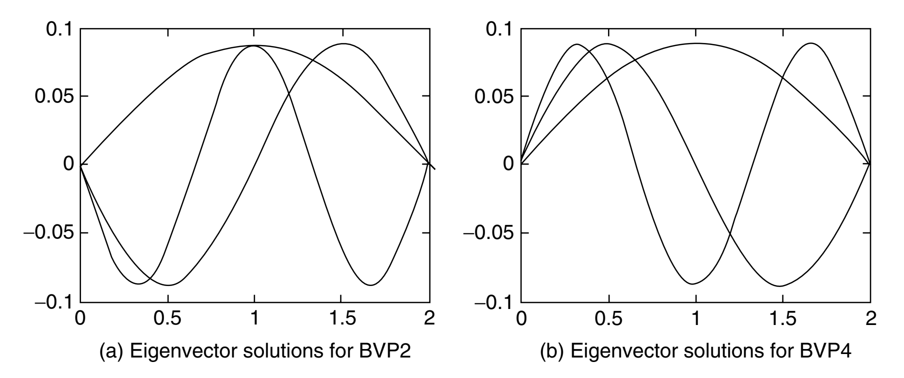


Figure P6.11 The eigenvector solutions of homogeneous second-order and fourth-order BVPs.

Now, use the routine "`bvp2_eig()`" with the number of grid points $N = 256$ to solve the BVP2 (P6.11.7) with $x_0 = 0$ and $x_f = 2$, find the lowest three angular frequencies (ω_i's) and plot the corresponding eigenvector solutions as depicted in Fig. P6.11a.

(b) A Homogeneous Fourth-Order BVP to an Eigenvalue Problem
Consider an eigenvalue boundary value problem of solving

$$\frac{d^4y}{dx^4} - \omega^4 y = 0 \tag{P6.11.9}$$

$$\text{with } y(x_0) = 0,\ \frac{d^2y}{dx^2}(x_0) = 0,\ y(x_f) = 0,\ \frac{d^2y}{dx^2}(x_f) = 0$$

to find $y(x)$ for $x \in [x_0, x_f]$ with the (possible) angular frequency ω.

In order to use the finite difference method, we divide the solution interval $[x_0, x_f]$ into N subintervals to have the grid points $x_i = x_0 + ih = x_0 + i(x_f - x_0)/N$ and then, replace the derivatives in the differential equation and the boundary conditions by their finite difference approximations to write

$$\frac{y_{i-2} - 4y_{i-1} + 6y_i - 4y_{i+1} + y_{i+2}}{h^4} - \omega^4 y_i = 0$$

$$y_{i-2} - 4y_{i-1} + 6y_i - 4y_{i+1} + y_{i+2} = \lambda y_i (\lambda = h^4\omega^4) \tag{P6.11.10}$$

with

$$y_0 = 0, \quad \frac{y_{-1} - 2y_0 + y_1}{h^2} = 0 \to y_{-1} = -y_1 \tag{P6.11.11a}$$

$$y_N = 0, \quad \frac{y_{N-1} - 2y_N + y_{N+1}}{h^2} = 0 \to y_{N+1} = -y_{N-1} \tag{P6.11.11b}$$

Substituting the discretized boundary condition (P6.11.11) into (P6.11.10) yields

$$y_{-1} - 4y_0 + 6y_1 - 4y_2 + y_3 = \lambda y_1 \xrightarrow{\text{(P6.11.11a)}}$$

$$5y_1 - 4y_2 + y_3 = \lambda y_1$$

$$y_0 - 4y_1 + 6y_2 - 4y_3 + y_4 = \lambda y_2 \xrightarrow{\text{(P6.11.11a)}}$$

$$-4y_1 + 6y_2 - 4y_3 + y_4 = \lambda y_2$$

$$y_i - 4y_{i+1} + 6y_{i+2} - 4y_{i+3} + y_{i+4} = \lambda y_{i+2}$$

$$\text{for } i = 1 : N-5 \tag{P6.11.12}$$

$$y_{N-4} - 4y_{N-3} + 6y_{N-2} - 4y_{N-1} + y_N = \lambda y_{N-2} \xrightarrow{\text{(P6.11.11b)}}$$

$$y_{N-4} - 4y_{N-3} + 6y_{N-2} - 4y_{N-1} = \lambda y_{N-2}$$

$$y_{N-3} - 4y_{N-2} + 6y_{N-1} - 4y_N + y_{N+1} = \lambda y_{N-1} \xrightarrow{\text{(P6.11.11b)}}$$

$$y_{N-3} - 4y_{N-2} + 5y_{N-1} = \lambda y_{N-1}$$

which can be formulated in a compact form as

$$\begin{bmatrix} 5 & -4 & 1 & 0 & 0 & 0 & 0 \\ -4 & 6 & -4 & 1 & 0 & 0 & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 \\ 0 & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\ 0 & 0 & 1 & -4 & 6 & -4 & 1 \\ 0 & 0 & 0 & 1 & -4 & 6 & -4 \\ 0 & 0 & 0 & 0 & 1 & -4 & 5 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ y_{N-3} \\ y_{N-2} \\ y_{N-1} \end{bmatrix} = \lambda \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ y_{N-3} \\ y_{N-2} \\ y_{N-1} \end{bmatrix}$$

$$A\mathbf{y} = \lambda \mathbf{y}, \qquad [A - \lambda I]\mathbf{y} = \mathbf{0} \tag{P6.11.13}$$

For this equation to have a nontrivial solution $\mathbf{y} \neq \mathbf{0}$, λ must be one of the eigenvalues of the matrix A and the corresponding eigenvectors are possible solutions. Note that the angular frequency corresponding to the eigenvalue λ can be obtained as

$$\omega = \sqrt[4]{\lambda}/h \tag{P6.11.14}$$

(i) Compose a routine "`bvp4_eig()`" which implements the above-mentioned scheme to solve the fourth-order eigenvalue problem (P6.11.9).

```
function [x,Y,ws,eigvals] = bvp4_eig(x0,xf,N)
```

(ii) Use the routine "`bvp4_eig()`" with the number of grid points $N = 256$ to solve the BVP4 (P6.11.9) with $x_0 = 0$ and $x_f = 2$, find the lowest three angular frequencies (ω_i's) and plot the corresponding eigenvector solutions as depicted in Fig. P6.11b.

(c) The Sturm–Liouville Equation

Consider an eigenvalue boundary value problem of solving

$$\frac{d}{dx}(f(x)y') + r(x)y = \lambda q(x)y \quad \text{with } y(x_0) = 0,\ y(x_f) = 0 \tag{P6.11.15}$$

to find $y(x)$ for $x \in [x_0, x_f]$ with the (possible) angular frequency ω.

In order to use the finite difference method, we divide the solution interval $[x_0, x_f]$ into N subintervals to have the grid points $x_i = x_0 + ih = x_0 + i(x_f - x_0)/N$, and then we replace the derivatives in the differential equation and the boundary conditions by their finite difference approximations (with the step size $h/2$) to write

$$\frac{f(x_i + h/2)y'(x_i + h/2) - f(x_i - h/2)y'(x_i - h/2)}{2(h/2)} + r(x_i)y_i = \lambda q(x_i)y(x_i)$$

$$\frac{1}{h}\left\{f\left(x_i + \frac{h}{2}\right)\frac{y_{i+1} - y_i}{h} - f\left(x_i - \frac{h}{2}\right)\frac{y_i - y_{i-1}}{h}\right\} + r(x_i)y_i = \lambda q(x_i)y(x_i)$$

$$a_i y_{i-1} + b_i y_i + c_i y_{i+1} = \lambda y_i \quad \text{for } i = 1, 2, \ldots, N-1 \tag{P6.11.16}$$

with

$$a_i = \frac{f(x_i - h/2)}{h^2 q(x_i)}, \quad c_i = \frac{f(x_i + h/2)}{h^2 q(x_i)}, \quad \text{and} \quad b_i = \frac{r(x_i)}{q(x_i)} - a_i - c_i \tag{P6.11.17}$$

(i) Compose a routine "`sturm()`" which implements the above-mentioned scheme to solve the Sturm–Liouville BVP (P6.11.15).

```
function [x,Y,ws,eigvals] = sturm(f,r,q,x0,xf,N)
```

(ii) Use the routine "`sturm()`" with the number of grid points $N = 256$ to solve the following BVP2:

$$\frac{d}{dx}\left((1 + x^2)y'\right) = -2\lambda y \quad \text{with} \quad y(x_0) = 0,\ y(x_f) = 0 \tag{P6.11.18}$$

Plot the eigenvector solutions corresponding to the lowest three angular frequencies (ω_i's).

7

OPTIMIZATION

Optimization involves finding the minimum/maximum of an objective function $f(x)$ subject to some constraint $x \in S$. If there is no constraint for x to satisfy—or, equivalently, S is the universe—then it is called an unconstrained optimization; otherwise, it is a constrained optimization. In this chapter, we will cover several unconstrained optimization techniques such as the golden search method, the quadratic approximation method, the Nelder–Mead method, the steepest descent method, the Newton method, the simulated-annealing (SA) method, and the genetic algorithm (GA). As for constrained optimization, we will only introduce the MATLAB built-in routines together with the routines for unconstrained optimization. Note that we don't have to distinguish maximization and minimization because maximizing $f(x)$ is equivalent to minimizing $-f(x)$ and so, without loss of generality, we deal only with the minimization problems.

7.1 UNCONSTRAINED OPTIMIZATION [L-2, CHAPTER 7]

7.1.1 Golden Search Method

This method is applicable to an unconstrained minimization problem such that the solution interval $[a, b]$ is known and the objective function $f(x)$ is unimodal within the interval; that is, the sign of its derivative $f'(x)$ changes at most once in $[a, b]$ so that $f(x)$ decreases/increases monotonically for $[a, x^o]/[x^o, b]$, where x^o is the solution that we are looking for. The so-called golden search procedure is summarized below and is cast into the routine "`opt_gs()`". We made a MATLAB

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

program "nm711.m", which uses this routine to find the minimum point of the objective function

$$f(x) = (x^2 - 4)^2/8 - 1 \tag{7.1.1}$$

GOLDEN SEARCH PROCEDURE

Step 1. Pick up the two points $c = a + (1 - r)h$ and $d = a + rh$ inside the interval $[a, b]$, where $r = (\sqrt{5} - 1)/2$ and $h = b - a$.

Step 2. If the values of $f(x)$ at the two points are almost equal [i.e., $f(a) \approx f(b)$] and the width of the interval is sufficiently small (i.e., $h \approx 0$), then stop the iteration to exit the loop and declare $x^o = c$ or $x^o = d$ depending on whether $f(c) < f(d)$ or not. Otherwise, go to Step 3.

Step 3. If $f(c) < f(d)$, let the new upper bound of the interval $b \leftarrow d$; otherwise, let the new lower bound of the interval $a \leftarrow c$. Then, go to Step 1.

```
function [xo,fo] = opt_gs(f,a,b,r,TolX,TolFun,k)
h = b - a; rh = r*h; c = b - rh; d = a + rh;
fc = feval(f,c); fd = feval(f,d);
if k <= 0 | (abs(h) < TolX & abs(fc - fd) < TolFun)
  if fc <= fd, xo = c;  fo = fc;
   else  xo = d; fo = fd;
  end
  if k == 0, fprintf('Just the best in given # of iterations'), end
else
  if fc < fd, [xo,fo] = opt_gs(f,a,d,r,TolX,TolFun,k - 1);
   else  [xo,fo] = opt_gs(f,c,b,r,TolX,TolFun,k - 1);
  end
end
```

```
%nm711.m  to perform the golden search method
f711 = inline('(x.*x-4).^2/8-1','x');
a = 0; b = 3; r =(sqrt(5)-1)/2; TolX = 1e-4; TolFun = 1e-4; MaxIter = 100;
[xo,fo] = opt_gs(f711,a,b,r,TolX,TolFun,MaxIter)
```

Figure 7.1 shows how the routine "opt_gs()" proceeds toward the minimum point step by step.

Note the following points about the golden search procedure.

- At every iteration, the new interval width is

$$b - c = b - (a + (1 - r)(b - a)) = rh \quad \text{or} \quad d - a = a + rh - a = rh \tag{7.1.2}$$

so that it becomes r times the old interval width ($b - a = h$).

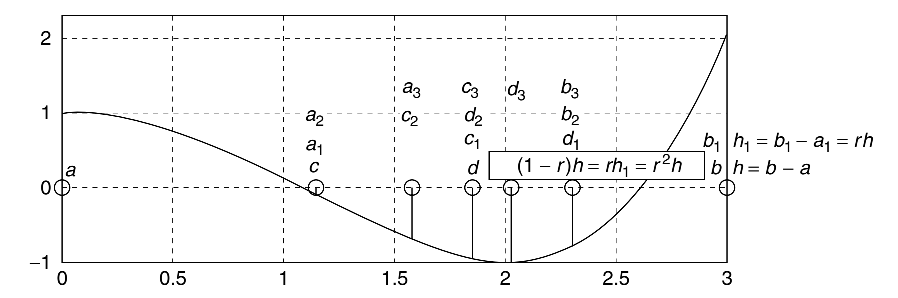


Figure 7.1 Process for the golden search method.

- The golden ratio r is fixed so that a point $c_1 = b_1 - rh_1 = b - r^2h$ in the new interval $[c, b]$ conforms with $d = a + rh = b - (1 - r)h$, that is,

$$r^2 = 1 - r, \quad r^2 + r - 1 = 0, \quad r = \frac{-1 + \sqrt{1+4}}{2} = \frac{-1 + \sqrt{5}}{2} \tag{7.1.3}$$

7.1.2 Quadratic Approximation Method

The idea of this method is to (a) approximate the objective function $f(x)$ by a quadratic function $p_2(x)$ matching the previous three (estimated solution) points and (b) keep updating the three points by replacing one of them with the minimum point of $p_2(x)$. More specifically, for the three points

$$\{(x_0, f_0), (x_1, f_1), (x_2, f_2)\} \qquad \text{with } x_0 < x_1 < x_2$$

we find the interpolation polynomial $p_2(x)$ of degree 2 to fit them and replace one of them with the zero of the derivative—that is, the root of $p_2'(x) = 0$ [see Eq. (P3.1.2) in Problem 3.1]:

$$x = x_3 = \frac{f_0(x_1^2 - x_2^2) + f_1(x_2^2 - x_0^2) + f_2(x_0^2 - x_1^2)}{2\{f_0(x_1 - x_2) + f_1(x_2 - x_0) + f_2(x_0 - x_1)\}} \tag{7.1.4}$$

In particular, if the previous estimated solution points are equidistant with an equal distance h (i.e., $x_2 - x_1 = x_1 - x_0 = h$), then this formula becomes

$$x_3 = \left.\frac{f_0(x_1^2 - x_2^2) + f_1(x_2^2 - x_0^2) + f_2(x_0^2 - x_1^2)}{2\{f_0(x_1 - x_2) + f_1(x_2 - x_0) + f_2(x_0 - x_1)\}}\right|_{\substack{x_1 = x_0 + h \\ x_2 = x_1 + h}}$$

$$= x_0 + h\frac{3f_0 - 4f_1 + f_2}{2(-f_0 + 2f_1 - f_2)} \tag{7.1.5}$$

We keep updating the three points this way until $|x_2 - x_0| \approx 0$ and/or $|f(x_2) - f(x_0)| \approx 0$, when we stop the iteration and declare x_3 as the minimum point. The rule for updating the three points is as follows.

1. In case $x_0 < x_3 < x_1$, we take $\{x_0, x_3, x_1\}$ or $\{x_3, x_1, x_2\}$ as the new set of three points depending on whether $f(x_3) < f(x_1)$ or not.
2. In case $x_1 < x_3 < x_2$, we take $\{x_1, x_3, x_2\}$ or $\{x_0, x_1, x_3\}$ as the new set of three points depending on whether $f(x_3) \leq f(x_1)$ or not.

This procedure, called the quadratic approximation method, is cast into the MATLAB routine "`opt_quad()`", which has the nested (recursive call) structure. We made the MATLAB program "`nm712.m`", which uses this routine to find the minimum point of the objective function (7.1.1) and also uses the MATLAB built-in routine "`fminbnd()`" to find it for cross-check. Figure 7.2 shows how the routine "`opt_quad()`" proceeds toward the minimum point step by step.

(cf) The MATLAB built-in routine "`fminbnd()`" corresponds to "`fmin()`" in the MATLAB of version.5.x.

```
function [xo,fo] = opt_quad(f,x0,TolX,TolFun,MaxIter)
%search for the minimum of f(x) by quadratic approximation method
if length(x0) > 2, x012 = x0(1:3);
 else
   if length(x0) == 2, a = x0(1); b = x0(2);
    else a = x0 - 10; b = x0 + 10;
   end
   x012 = [a (a + b)/2 b];
end
f012 = f(x012);
[xo,fo] = opt_quad0(f,x012,f012,TolX,TolFun,MaxIter);
```

```
function [xo,fo] = opt_quad0(f,x012,f012,TolX,TolFun,k)
x0 = x012(1); x1 = x012(2); x2 = x012(3);
f0 = f012(1); f1 = f012(2); f2 = f012(3);
nd = [f0 - f2 f1 - f0 f2 - f1]*[x1*x1 x2*x2 x0*x0; x1 x2 x0]';
x3 = nd(1)/2/nd(2); f3 = feval(f,x3); %Eq.(7.1.4)
if k <= 0 | abs(x3 - x1) < TolX | abs(f3 - f1) < TolFun
  xo = x3;  fo = f3;
  if k == 0, fprintf('Just the best in given # of iterations'), end
 else
  if x3 < x1
    if f3 < f1, x012 = [x0 x3 x1]; f012 = [f0 f3 f1];
     else  x012 = [x3 x1 x2]; f012 = [f3 f1 f2];
    end
   else
    if f3 <= f1, x012 = [x1 x3 x2]; f012 = [f1 f3 f2];
     else  x012 = [x0 x1 x3]; f012 = [f0 f1 f3];
    end
  end
  [xo,fo] = opt_quad0(f,x012,f012,TolX,TolFun,k - 1);
end
```

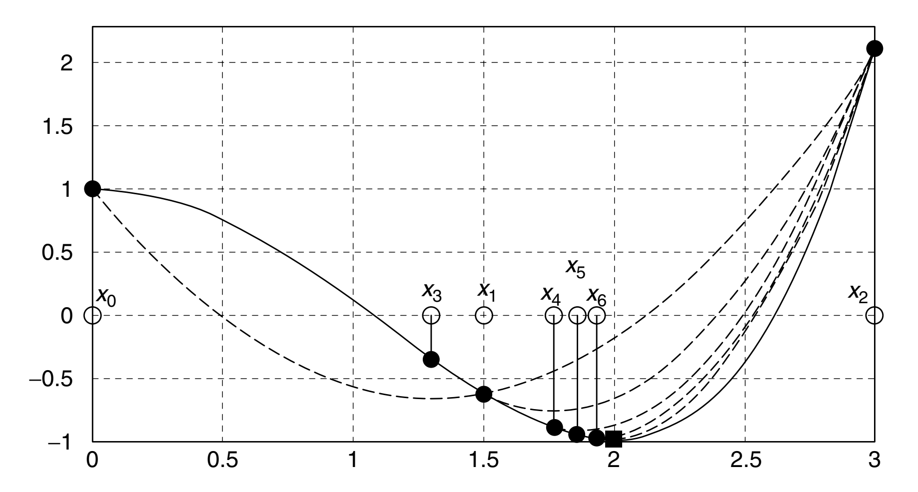


Figure 7.2 Process of searching for the minimum by the quadratic approximation method.

```
%nm712.m  to perform the quadratic approximation method
clear, clf
f711 = inline('(x.*x - 4).^2/8-1', 'x');
a = 0; b = 3; TolX = 1e-5; TolFun = 1e-8; MaxIter = 100;
[xoq,foq] = opt_quad(f711,[a b],TolX,TolFun,MaxIter)
%minimum point and its function value
[xob,fob] = fminbnd(f711,a,b) %MATLAB built-in function
```

7.1.3 Nelder–Mead Method [W-8]

The Nelder–Mead method is applicable to the minimization of a multivariable objective function, for which neither the golden search method nor the quadratic approximation method can be applied. The algorithm of the Nelder–Mead method summarized in the box below is cast into the MATLAB routine "`Nelder0()`". Note that in the N-dimensional case ($N > 2$), this algorithm should be repeated for each subplane as implemented in the outer routine "`opt_Nelder()`".

We made the MATLAB program "`nm713.m`" to minimize a two-variable objective function

$$f(x_1, x_2) = x_1^2 - x_1x_2 - 4x_1 + x_2^2 - x_2 \tag{7.1.6}$$

whose minimum can be found in an analytical way—that is, by setting the partial derivatives of $f(x_1, x_2)$ with respect to x_1 and x_2 to zero as

$$\left.\begin{aligned} \frac{\partial}{\partial x_1} f(x_1, x_2) = 2x_1 - x_2 - 4 = 0 \\ \frac{\partial}{\partial x_2} f(x_1, x_2) = 2x_2 - x_1 - 1 = 0 \end{aligned}\right\} \qquad \mathbf{x}_o = (x_{1o}, x_{2o}) = (3, 2)$$

NELDER–MEAD ALGORITHM

Step 1. Let the initial three estimated solution points be a, b and c, where $f(a) < f(b) < f(c)$.

Step 2. If the three points or their function values are sufficiently close to each other, then declare a to be the minimum and terminate the procedure.

Step 3. Otherwise, expecting that the minimum we are looking for may be at the opposite side of the worst point c over the line $\overline{ab}$ (see Fig. 7.3), take

$$e = m + 2(m - c), \qquad \text{where } m = (a + b)/2$$

and if $f(e) < f(b)$, take e as the new c; otherwise, take

$$r = (m + e)/2 = 2m - c$$

and if $f(r) < f(c)$, take r as the new c; if $f(r) \geq f(b)$, take

$$s = (c + m)/2$$

and if $f(s) < f(c)$, take s as the new c; otherwise, give up the two points b,c and take m and $c_1 = (a + c)/2$ as the new b and c, reflecting our expectation that the minimum would be around a.

Step 4. Go back to Step 1.

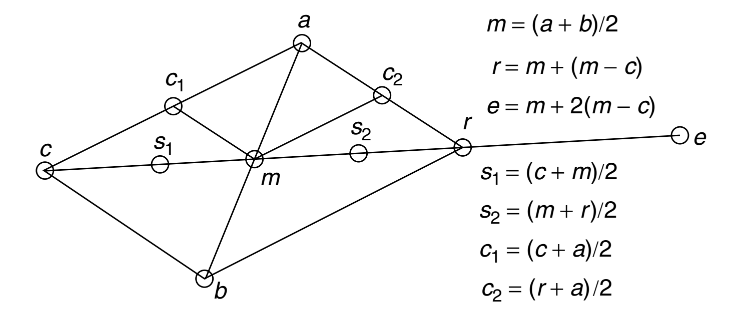


Figure 7.3 Notation used in the Nelder–Mead method.

```
function [xo,fo] = Nelder0(f,abc,fabc,TolX,TolFun,k)
[fabc,I] = sort(fabc); a = abc(I(1),:); b = abc(I(2),:); c = abc(I(3),:);
fa = fabc(1); fb = fabc(2); fc = fabc(3);  fba = fb - fa; fcb = fc - fb;
if k <= 0 | abs(fba) + abs(fcb) < TolFun | abs(b - a) + abs(c - b) < TolX
  xo = a; fo = fa;
  if k == 0, fprintf('Just best in given # of iterations'), end
 else
  m = (a + b)/2;  e = 3*m - 2*c;  fe = feval(f,e);
  if fe < fb, c = e;  fc = fe;
   else
     r = (m+e)/2;  fr = feval(f,r);
     if fr < fc,  c = r;  fc = fr;  end
     if fr >= fb
       s = (c + m)/2;  fs = feval(f,s);
       if fs < fc,  c = s;  fc = fs;
        else b = m; c = (a + c)/2;  fb = feval(f,b); fc = feval(f,c);
       end
     end
  end
  [xo,fo] = Nelder0(f,[a;b;c],[fa fb fc],TolX,TolFun,k - 1);
end
```

```
function [xo,fo] = opt_Nelder(f,x0,TolX,TolFun,MaxIter)
N = length(x0);
if N == 1 %for 1-dimensional case
  [xo,fo] = opt_quad(f,x0,TolX,TolFun); return
end
S = eye(N);
for i = 1:N %repeat the procedure for each subplane
   i1 = i + 1; if i1 > N, i1 = 1; end
   abc = [x0; x0 + S(i,:); x0 + S(i1,:)]; %each directional subplane
   fabc = [feval(f,abc(1,:)); feval(f,abc(2,:)); feval(f,abc(3,:))];
   [x0,fo] = Nelder0(f,abc,fabc,TolX,TolFun,MaxIter);
   if N < 3, break; end %No repetition needed for a 2-dimensional case
end
xo = x0;
```

```
%nm713.m: do_Nelder
f713 = inline('x(1)*(x(1)-4-x(2)) +x(2)*(x(2)-1)','x');
x0 = [0 0], TolX = 1e-4; TolFun = 1e-9; MaxIter = 100;
[xon,fon] = opt_Nelder(f713,x0,TolX,TolFun,MaxIter)
%minimum point and its function value
[xos,fos] = fminsearch(f713,x0) %use the MATLAB built-in function
```

This program also applies the MATLAB built-in routine "`fminsearch()`" to minimize the same objective function for practice and confirmation. The minimization process is illustrated in Fig. 7.4.

(cf) The MATLAB built-in routine "`fminsearch()`" uses the Nelder–Mead algorithm to minimize a multivariable objective function. It corresponds to "`fmins()`" in the MATLAB of version.5.x.

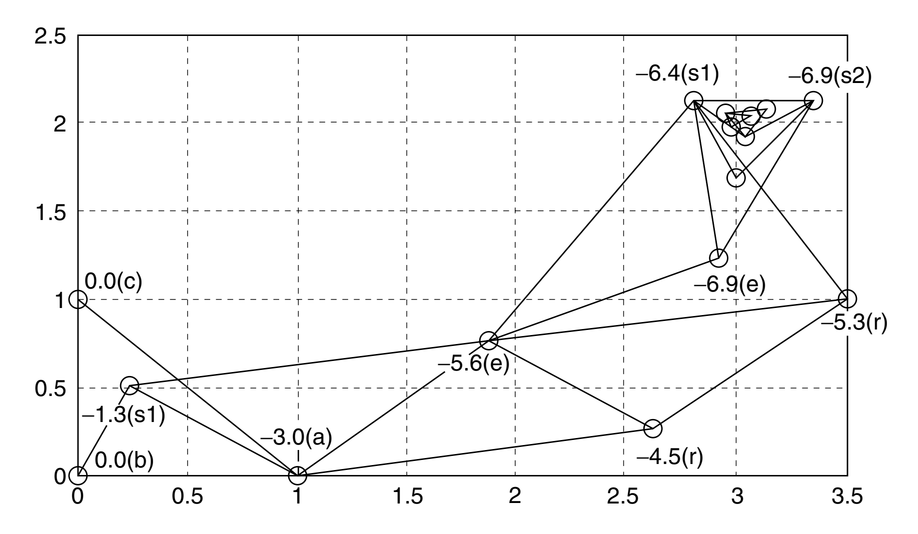


Figure 7.4 Process for the Nelder–Mead method (`nm713.m-opt_Nelder()`).

7.1.4 Steepest Descent Method

This method searches for the minimum of an N-dimensional objective function in the direction of a negative gradient

$$-\mathbf{g}(\mathbf{x}) = -\nabla f(\mathbf{x}) = -\left[\frac{\partial f(\mathbf{x})}{\partial x_1} \ \frac{\partial f(\mathbf{x})}{\partial x_2} \ \cdots \ \frac{\partial f(\mathbf{x})}{\partial x_N}\right]^T \tag{7.1.7}$$

with the step-size α_k (at iteration k) adjusted so that the function value is minimized along the direction by a (one-dimensional) line search technique like the quadratic approximation method. The algorithm of the steepest descent method is summarized in the following box and cast into the MATLAB routine “`opt_steep()`”.

We made the MATLAB program “`nm714.m`” to minimize the objective function (7.1.6) by using the steepest descent method. The minimization process is illustrated in Fig. 7.5.

STEEPEST DESCENT ALGORITHM

Step 0. With the iteration number $k = 0$, find the function value $f_0 = f(\mathbf{x}_0)$ for the initial point $\mathbf{x}_0$.

Step 1. Increment the iteration number k by one, find the step-size α_{k-1} along the direction of the negative gradient $-\mathbf{g}_{k-1}$ by a (one-dimensional) line

search like the quadratic approximation method.

$$\alpha_{k-1} = \text{ArgMin}_{\alpha} f(\mathbf{x}_{k-1} - \alpha \mathbf{g}_{k-1}/\|\mathbf{g}_{k-1}\|) \tag{7.1.8}$$

Step 2. Move the approximate minimum by the step-size α_{k-1} along the direction of the negative gradient $-\mathbf{g}_{k-1}$ to get the next point

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \alpha_{k-1}\mathbf{g}_{k-1}/\|\mathbf{g}_{k-1}\| \tag{7.1.9}$$

Step 3. If $\mathbf{x}_k \approx \mathbf{x}_{k-1}$ and $f(\mathbf{x}_k) \approx f(\mathbf{x}_{k-1})$, then declare $\mathbf{x}_k$ to be the minimum and terminate the procedure. Otherwise, go back to step 1.

```
function [xo,fo] = opt_steep(f,x0,TolX,TolFun,alpha0,MaxIter)
% minimize the ftn f by the steepest descent method.
%input:  f  = ftn to be given as a string 'f'
%        x0 = the initial guess of the solution
%output: x0 = the minimum point reached
%        f0 = f(x(0))
if nargin < 6, MaxIter = 100; end %maximum # of iteration
if nargin < 5, alpha0 = 10; end %initial step size
if nargin < 4, TolFun = 1e-8; end %|f(x)| < TolFun wanted
if nargin < 3, TolX = 1e-6; end %|x(k)- x(k - 1)|<TolX wanted
x = x0;  fx0 = feval(f,x0); fx = fx0;
alpha = alpha0;  kmax1 = 25;
warning = 0; %the # of vain wanderings to find the optimum step size
for k = 1: MaxIter
   g = grad(f,x); g = g/norm(g); %gradient as a row vector
   alpha = alpha*2; %for trial move in negative gradient direction
   fx1  = feval(f,x - alpha*2*g);
   for k1 = 1:kmax1 %find the optimum step size(alpha) by line search
      fx2 = fx1; fx1 = feval(f,x-alpha*g);
      if fx0 > fx1+TolFun & fx1 < fx2 - TolFun %fx0 > fx1 < fx2
        den = 4*fx1 - 2*fx0 - 2*fx2; num = den - fx0 + fx2; %Eq.(7.1.5)
        alpha = alpha*num/den;
        x = x - alpha*g;  fx = feval(f,x); %Eq.(7.1.9)
        break;
       else  alpha = alpha/2;
      end
   end
   if k1 >= kmax1, warning = warning + 1; %failed to find optimum step size
    else  warning = 0;
   end
   if warning >= 2|(norm(x - x0) < TolX&abs(fx - fx0) < TolFun), break;  end
   x0 = x;  fx0 = fx;
end
xo = x;  fo = fx;
if k == MaxIter, fprintf('Just best in %d iterations',MaxIter), end
```

```
%nm714
f713 = inline('x(1)*(x(1) - 4 - x(2)) + x(2)*(x(2)- 1)','x');
x0 = [0 0], TolX = 1e-4; TolFun = 1e-9; alpha0 = 1; MaxIter = 100;
[xo,fo] = opt_steep(f713,x0,TolX,TolFun,alpha0,MaxIter)
```

7.1.5 Newton Method

Like the steepest descent method, this method also uses the gradient to search for the minimum point of an objective function. Such gradient-based optimization methods are supposed to reach a point at which the gradient is (close to) zero. In this context, the optimization of an objective function $f(\mathbf{x})$ is equivalent to finding a zero of its gradient $\mathbf{g}(\mathbf{x})$, which in general is a vector-valued function of a vector-valued independent variable $\mathbf{x}$. Therefore, if we have the gradient function $\mathbf{g}(\mathbf{x})$ of the objective function $f(\mathbf{x})$, we can solve the system of nonlinear equations $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ to get the minimum of $f(\mathbf{x})$ by using the Newton method explained in Section 4.4.

The backgrounds of this method as well as the steepest descent method can be shown by taking the Taylor series of, say, a two-variable objective function $f(x_1, x_2)$:

$$f(x_1, x_2) \cong f(x_{1k}, x_{2k}) + \begin{bmatrix} \dfrac{\partial f}{\partial x_1} & \dfrac{\partial f}{\partial x_2} \end{bmatrix}_{(x_{1k}, x_{2k})} \begin{bmatrix} x_1 - x_{1k} \\ x_2 - x_{2k} \end{bmatrix}$$

$$+ \frac{1}{2} \begin{bmatrix} x_1 - x_{1k} & x_2 - x_{2k} \end{bmatrix} \begin{bmatrix} \partial^2 f/\partial x_1^2 & \partial^2 f/\partial x_1 \partial x_2 \\ \partial^2 f/\partial x_2 \partial x_1 & \partial^2 f/\partial x_2^2 \end{bmatrix} \Bigg|_{(x_{1k}, x_{2k})} \begin{bmatrix} x_1 - x_{1k} \\ x_2 - x_{2k} \end{bmatrix}$$

$$f(\mathbf{x}) \cong f(\mathbf{x}_k) + \nabla f(\mathbf{x})^T|_{\mathbf{x}_k}[\mathbf{x} - \mathbf{x}_k] + \frac{1}{2}[\mathbf{x} - \mathbf{x}_k]^T \nabla^2 f(\mathbf{x})|_{\mathbf{x}_k}[\mathbf{x} - \mathbf{x}_k]$$

$$f(\mathbf{x}) \cong f(\mathbf{x}_k) + \mathbf{g}_k^T[\mathbf{x} - \mathbf{x}_k] + \frac{1}{2}[\mathbf{x} - \mathbf{x}_k]^T H_k[\mathbf{x} - \mathbf{x}_k] \tag{7.1.10}$$

with the gradient vector $\mathbf{g}_k = \nabla f(\mathbf{x})|_{\mathbf{x}_k}$ and the Hessian matrix $H_k = \nabla^2 f(\mathbf{x})|_{\mathbf{x}_k}$. In the light of this equation, we can see that the value of the objective function at point $\mathbf{x}_{k+1}$ updated by the steepest descent algorithm described by Eq. (7.1.9)

$$\mathbf{x}_{k+1} \overset{(7.1.9)}{=} \mathbf{x}_k - \alpha_k \mathbf{g}_k / \|\mathbf{g}_k\|$$

is most likely smaller than that at the old point $\mathbf{x}_k$, with the third term in Eq. (7.1.10) neglected.

$$\begin{aligned} f(\mathbf{x}_{k+1}) &\cong f(\mathbf{x}_k) + \mathbf{g}_k^T[\mathbf{x}_{k+1} - \mathbf{x}_k] = f(\mathbf{x}_k) - \alpha_k \mathbf{g}_k^T \mathbf{g}_k / \|\mathbf{g}_k\| \\ f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k) &\cong -\alpha_k \mathbf{g}_k^T \mathbf{g}_k / \|\mathbf{g}_k\| \le 0 \Rightarrow f(\mathbf{x}_{k+1}) \le f(\mathbf{x}_k) \end{aligned} \tag{7.1.11}$$

Slightly different from this strategy of the steepest descent algorithm, the Newton method tries to go straight to the zero of the gradient of the approximate objective function (7.1.10)

$$\mathbf{g}_k + H_k[\mathbf{x} - \mathbf{x}_k] = \mathbf{0}, \qquad \mathbf{x} = \mathbf{x}_k - H_k^{-1}\mathbf{g}_k \tag{7.1.12}$$

by the updating rule

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H_k^{-1}\mathbf{g}_k \tag{7.1.13}$$

with the gradient vector $\mathbf{g}_k = \nabla f(\mathbf{x})|_{\mathbf{x}_k}$ and the Hessian matrix $H_k = \nabla^2 f(\mathbf{x})|_{\mathbf{x}_k}$ (Appendix C).

This algorithm is essentially to find the zero of the gradient function $\mathbf{g}(\mathbf{x})$ of the objective function and consequently, it can be implemented by using any vector nonlinear equation solver. What we have to do is just to define the gradient function $\mathbf{g}(\mathbf{x})$ and put the function name as an input argument of any routine like "`newtons()`" or "`fsolve()`" for solving a system of nonlinear equations (see Section 4.6).

Now, we make a MATLAB program "`nm715.m`", which actually solves $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ for the gradient function

$$\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1} \ \frac{\partial f}{\partial x_2}\right]^T = \left[\, 2x_1 - x_2 - 4 \quad 2x_2 - x_1 - 1 \,\right] \tag{7.1.14}$$

of the objective function (7.1.6)

$$f(\mathbf{x}) = f(x_1, x_2) = x_1^2 - x_1x_2 - 4x_1 + x_2^2 - x_2$$

Figure 7.5 illustrates the process of searching for the minimum point by the Newton algorithm (7.1.13) as well as the steepest descent algorithm (7.1.9), where the steepest descent algorithm proceeds in the negative gradient direction until the minimum point in the line is reached, while the Newton algorithm approaches the minimum point almost straightly and reaches it in a few iterations.

```
>>nm715
   xo = [3.0000  2.0000], ans = -7
```

```
%nm715 to minimize an objective ftn f(x) by the Newton method.
clear, clf
f713 = inline('x(1).^2 - 4*x(1) - x(1).*x(2) + x(2).^2 - x(2)','x');
g713 = inline('[2*x(1) - x(2) - 4 2*x(2) - x(1) - 1]','x');
x0 = [0 0],  TolX = 1e-4; TolFun = 1e-6;  MaxIter = 50;
[xo,go,xx] = newtons(g713,x0,TolX,MaxIter);
xo, f713(xo) %an extremum point reached and its function value
```

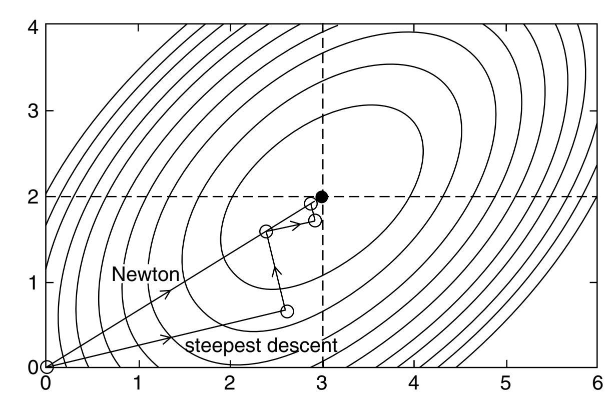


Figure 7.5 Process for the steepest descent method and Newton method ("`nm714.m`" and "`nm715.m`").

Remark 7.1. Weak Point of Newton Method.

The Newton method is usually more efficient than the steepest descent method if only it works as illustrated above, but it is not guaranteed to reach the minimum point. The decisive weak point of the Newton method is that it may approach one of the extrema having zero gradient, which is not necessarily a (local) minimum, but possibly a maximum or a saddle point (see Fig. 7.13).

7.1.6 Conjugate Gradient Method

Like the steepest descent method or Newton method, this method also uses the gradient to search for the minimum point of an objective function, but in a different way. It has two versions—the Polak–Ribiere (PR) method and the Fletcher–Reeves (FR) method—that are slightly different only in the search direction vector. This algorithm, summarized in the following box, is cast into the MATLAB routine "`opt_conjg()`", which implements PR or FR depending on the last input argument `KC = 1` or 2. The quasi-Newton algorithm used in the MATLAB built-in routine "`fminunc()`" is similar to the conjugate gradient method.

This method borrows the framework of the steepest descent method and needs a bit more effort for computing the search direction vector $\mathbf{s}(n)$. It takes at most N iterations to reach the minimum point in case the objective function is quadratic with a positive-definite Hessian matrix H as

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{b}^T\mathbf{x} + c \quad \text{where } \mathbf{x}\text{: an } N\text{-dimensional vector} \tag{7.1.15}$$

CONJUGATE GRADIENT ALGORITHM

Step 0. With the iteration number $k = 0$, find the objective function value $f_0 = f(\mathbf{x}_0)$ for the initial point $\mathbf{x}_0$.

Step 1. Initialize the inside loop index, the temporary solution and the search direction vector to $n = 0$, $\mathbf{x}(n) = \mathbf{x}_k$ and $\mathbf{s}(n) = -\mathbf{g}_k = -\mathbf{g}(\mathbf{x}_k)$, respectively, where $\mathbf{g}(\mathbf{x})$ is the gradient of the objective function $f(\mathbf{x})$.

Step 2. For $n = 0$ to $N - 1$, repeat the following things:

Find the (optimal) step-size

$$\alpha_n = \text{ArgMin}_\alpha f(\mathbf{x}(n) + \alpha\mathbf{s}(n)) \tag{7.1.16}$$

and update the temporary solution point to

$$\mathbf{x}(n+1) = \mathbf{x}(n) + \alpha_n\mathbf{s}(n) \tag{7.1.17}$$

and the search direction vector to

$$\mathbf{s}(n+1) = -\mathbf{g}_{n+1} + \beta_n\mathbf{s}(n) \tag{7.1.18}$$

with

$$\beta_n = \frac{[\mathbf{g}_{n+1} - \mathbf{g}_n]^T \mathbf{g}_{n+1}}{\mathbf{g}_n^T \mathbf{g}_n} \text{(FR)} \quad \text{or} \quad \frac{\mathbf{g}_{n+1}^T \mathbf{g}_{n+1}}{\mathbf{g}_n^T \mathbf{g}_n} \text{(PR)} \tag{7.1.19}$$

Step 3. Update the approximate solution point to $\mathbf{x}_{k+1} = \mathbf{x}(N)$, which is the last temporary one.

Step 4. If $\mathbf{x}_k \approx \mathbf{x}_{k-1}$ and $f(\mathbf{x}_k) \approx f(\mathbf{x}_{k-1})$, then declare $\mathbf{x}_k$ to be the minimum and terminate the procedure. Otherwise, increment k by one and go back to Step 1.

```
function [xo,fo] = opt_conjg(f,x0,TolX,TolFun,alpha0,MaxIter,KC)
%KC = 1: Polak–Ribiere Conjugate Gradient method
%KC = 2: Fletcher–Reeves Conjugate Gradient method
if nargin < 7, KC = 0; end
if nargin < 6, MaxIter = 100; end
if nargin < 5, alpha0 = 10; end
if nargin < 4, TolFun = 1e-8; end
if nargin < 3, TolX = 1e-6; end
N = length(x0); nmax1 = 20; warning = 0; h = 1e-4; %dimension of variable
x = x0;  fx = feval(f,x0);  fx0 = fx;
for k = 1: MaxIter
  xk0 = x; fk0 = fx; alpha = alpha0;
  g = grad(f,x,h);  s = -g;
  for n = 1:N
     alpha = alpha0;
     fx1 = feval(f,x + alpha*2*s); %trial move in search direction
     for n1 = 1:nmax1 %To find the optimum step size by line search
        fx2 = fx1; fx1 = feval(f,x+alpha*s);
        if fx0 > fx1 + TolFun & fx1 < fx2 - TolFun %fx0 > fx1 < fx2
          den = 4*fx1 - 2*fx0 - 2*fx2; num = den-fx0 + fx2; %Eq.(7.1.5)
          alpha = alpha*num/den;
          x = x+alpha*s;  fx = feval(f,x);
          break;
         elseif n1 == nmax1/2
          alpha = -alpha0;  fx1 = feval(f,x + alpha*2*s);
         else
          alpha = alpha/2;
        end
     end
     x0 = x;  fx0 = fx;
     if n < N
       g1 = grad(f,x,h);
       if KC <= 1,  s = - g1 +(g1 - g)*g1'/(g*g'+ 1e-5)*s; %(7.1.19a)
        else  s = -g1 + g1*g1'/(g*g'+ 1e-5)*s; %(7.1.19b)
       end
       g = g1;
     end
     if n1 >= nmax1, warning = warning+1; %can't find optimum step size
      else  warning = 0;
     end
  end
  if warning >= 2|(norm(x - xk0)<TolX&abs(fx - fk0)< TolFun), break;  end
end
xo = x;  fo = fx;
if k == MaxIter, fprintf('Just best in %d iterations',MaxIter), end
```

```
%nm716  to minimize f(x) by the conjugate gradient method.
f713 = inline('x(1).^2 - 4*x(1) - x(1).*x(2) + x(2).^2 - x(2)','x');
x0 =[0 0], TolX = 1e-4; TolFun = 1e-4; alpha0 = 10; MaxIter = 100;
[xo,fo] = opt_conjg(f713,x0,TolX,TolFun,alpha0,MaxIter,1)
[xo,fo] = opt_conjg(f713,x0,TolX,TolFun,alpha0,MaxIter,2)
```

Based on the fact that minimizing this quadratic objective function is equivalent to solving the linear equation

$$\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x}) = H\mathbf{x} + \mathbf{b} = \mathbf{0} \tag{7.1.20}$$

MATLAB has several built-in routines such as "`cgs()`","`pcg()`", and "`bicg()`", which use the conjugate gradient method to solve a set of linear equations.

We make the MATLAB program "`nm716.m`" to minimize the objective function (7.1.6) by the conjugate gradient method and the minimization process is illustrated in Fig. 7.6.

7.1.7 Simulated Annealing Method [W-7]

All of the optimization methods discussed so far may be more or less efficient in finding the minimum point if only they start from the initial point sufficiently close to it. But, the point they reach may be one of several local minima and we often cannot be sure that it is the global minimum. How about repeating the procedure to search for all local minima starting from many different initial guesses and taking the best one as the global minimum? This would be a computationally formidable task, since there is no systematic way to determine a suitable sequence of initial guesses, each of which leads to its own (local) minimum so that all the local minima can be exhaustively found to compete with each other for the global minimum.

An interesting alternative is based on the analogy between annealing and minimization. Annealing is the physical process of heating up a solid metal above its melting point and then cooling it down so slowly that the highly excited atoms can settle into a (global) minimum energy state, yielding a single crystal with a regular structure. Fast cooling by rapid quenching may result in widespread

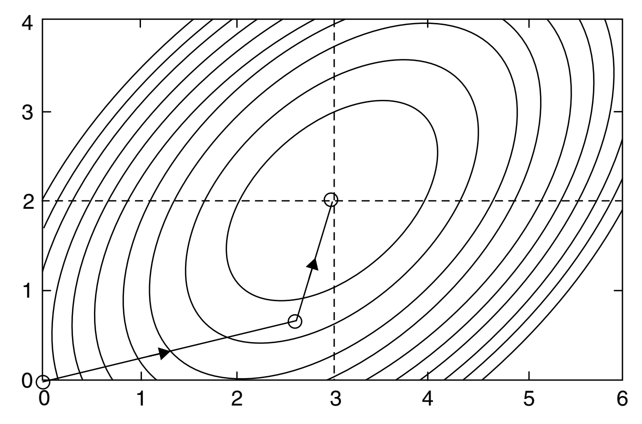


Figure 7.6 Process for the conjugate gradient method ("`nm716.m`").

irregularities and defects in the crystal structure, analogous to being too hasty to find the global minimum. The simulated annealing process can be implemented using the Boltzmann probability distribution of an energy level $E(\geq 0)$ at temperature T described by

$$p(E) = \alpha \exp(-E/KT) \quad \text{with the Boltzmann constant } K \text{ and } \alpha = 1/KT \tag{7.1.21}$$

Note that at high temperature the probability distribution curve is almost flat over a wide range of E, implying that the system can be in a high energy state as equally well as in a low energy state, while at low temperature the probability distribution curve gets higher/lower for lower/higher E, implying that the system will most probably be in a low energy state, but still have a slim chance to be in a high energy state so that it can escape from a local minimum energy state.

The idea of simulated annealing is summarized in the box below and cast into the MATLAB routine "`sim_anl()`". This routine has two parts that vary with the iteration number as the temperature falls down. One is the size of step $\Delta\mathbf{x}$ from the previous guess to the next guess, which is made by generating a random vector $\mathbf{y}$ having uniform distribution $U[-1, +1]$ and the same dimension as the variable $\mathbf{x}$ and multiplying $\mu^{-1}(\mathbf{y})$ (in a termwise manner) by the difference vector $(\mathbf{u} - \mathbf{l})$ between the upper bound $\mathbf{u}$ and the lower bound $\mathbf{l}$ of the domain of $\mathbf{x}$. The μ^{-1}-law

$$g_\mu^{-1}(y) = \frac{(1+\mu)^{|y|} - 1}{\mu} \operatorname{sign}(y) \qquad \text{for } |y| \leq 1 \tag{7.1.22}$$

implemented in the routine "`mu_inv()`" has the parameter μ that is increased according to a rule

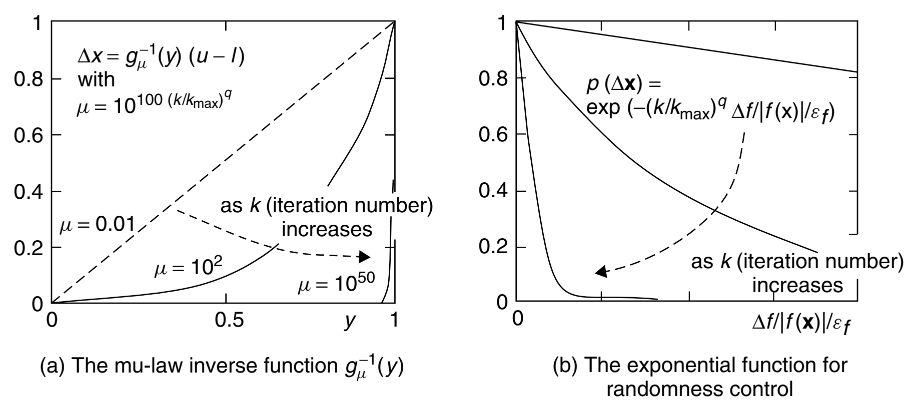
$$\mu = 10^{100\,(k/k_{\max})^q} \qquad \text{with } q > 0\text{: the quenching factor} \tag{7.1.23}$$

as the iteration number k increases, reaching $\mu = 10^{100}$ at the last iteration $k = k_{\max}$. Note the following:

- The quenching factor $q > 0$ is made small/large for slow/fast quenching.
- The value of μ^{-1}-law function becomes small for $|y| < 1$ as μ increases (see Fig. 7.7a).

The other is the probability of taking a step $\Delta\mathbf{x}$ that would result in change $\Delta f > 0$ of the objective function value $f(\mathbf{x})$. Similarly to Eq. (7.1.21), this is determined by

$$p(\text{taking the step } \Delta\mathbf{x}) = \exp\left(-\left(\frac{k}{k_{\max}}\right)^q \frac{\Delta f}{|f(\mathbf{x})|\varepsilon_f}\right) \quad \text{for } \Delta f > 0 \tag{7.1.24}$$



(a) The mu-law inverse function $g_\mu^{-1}(y)$

(b) The exponential function for randomness control

Figure 7.7 Illustrative functions used for controlling the randomness–temperature in SA.

SIMULATED ANNEALING

Step 0. Pick the initial guess $\mathbf{x}_0$, the lower bound $\mathbf{l}$, the upper bound $\mathbf{u}$, the maximum number of iterations $k_{\max} > 0$, the quenching factor $q > 0$ (to be made small/large for slow/fast quenching), and the relative tolerance ε_f of function value fluctuation.

Step 1. Let $\mathbf{x} = \mathbf{x}_0$, $\mathbf{x}^o = \mathbf{x}$, $f^o = f(\mathbf{x})$.

Step 2. For $k = 1$ to $k_{\max}$, do

{Generate an $N \times 1$ uniform random vector of $U[-1, +1]$ and transform it by the inverse μ law (with $\mu = 10^{100\,(k/k_{\max})^q}$) to make $\Delta\mathbf{x}$ and then take $\mathbf{x}_1 \leftarrow \mathbf{x} + \Delta\mathbf{x}$, confining the next guess inside the admissible region $\{\mathbf{x}|\mathbf{l} \le \mathbf{x} \le \mathbf{u}\}$ as needed.

If $\Delta f = f(\mathbf{x}_1) - f(\mathbf{x}) < 0$,

$$\{\text{set } \mathbf{x} \leftarrow \mathbf{x}_1 \text{ and if } f(\mathbf{x}) < f^o, \text{set } \mathbf{x}^o \leftarrow \mathbf{x} \text{ and } f^o \leftarrow f(\mathbf{x}^o).\}$$

Otherwise,

{generate a uniform random number z of U[0,1] and set $\mathbf{x} \leftarrow \mathbf{x}_1$ only in case

$$z < p(\text{taking the step } \Delta\mathbf{x}) \overset{(7.1.24)}{=} \exp(-(k/k_{\max})^q \Delta f/|f(\mathbf{x})|/\varepsilon_f)$$

}

}

Step 3. Regarding $\mathbf{x}^o$ as close to the minimum point that we are looking for, we may set $\mathbf{x}^o$ as the initial value and apply any (local) optimization algorithm to search for the minimum point of $f(\mathbf{x})$.

```
function [xo,fo] = sim_anl(f,x0,l,u,kmax,q,TolFun)
% simulated annealing method to minimize f(x) s.t. l <= x <= u
N = length(x0);
x = x0; fx = feval(f,x);
xo = x; fo = fx;
if nargin < 7, TolFun = 1e-8; end
if nargin < 6, q = 1; end %quenching factor
if nargin < 5, kmax = 100; end %maximum iteration number
for k = 0:kmax
   Ti = (k/kmax)^q; %inverse of temperature from 0 to 1
   mu = 10^(Ti*100); % Eq.(7.1.23)
   dx = mu_inv(2*rand(size(x))- 1,mu).*(u - l);
   x1 = x + dx; %next guess
   x1 = (x1 < l).*l +(l <= x1).*(x1 <= u).*x1 +(u < x1).*u;
    %confine it inside the admissible region bounded by l and u.
   fx1 = feval(f,x1); df = fx1- fx;
   if df < 0|rand < exp(-Ti*df/(abs(fx) + eps)/TolFun) Eq.(7.1.24)
     x = x1; fx = fx1;
   end
   if fx < fo, xo = x; fo = fx1; end
end
```

```
function x = mu_inv(y,mu) % inverse of mu-law Eq.(7.1.22)
x = (((1+mu).^abs(y)- 1)/mu).*sign(y);
```

```
%nm717 to minimize an objective function f(x) by various methods.
clear, clf
f = inline('x(1)^4 - 16*x(1)^2 - 5*x(1) + x(2)^4 - 16*x(2)^2 - 5*x(2)','x');
l = [-5 -5]; u = [5 5]; %lower/upperbound
x0 = [0 0]
[xo_nd,fo] = opt_Nelder(f,x0)
[xos,fos] = fminsearch(f,x0) %cross-check by MATLAB built-in routines
[xou,fou] = fminunc(f,x0)
kmax = 500; q = 1; TolFun = 1e-9;
[xo_sa,fo_sa] = sim_anl(f,x0,l,u,kmax,q,TolFun)
```

which remains as big as e^{-1} for $|\Delta f/f(\mathbf{x})| = \varepsilon_f$ at the last iteration $k = k_{\max}$, meaning that the probability of taking a step hopefully to escape from a local minimum and find the global minimum at the risk of increasing the value of objective function by the amount $\Delta f = |f(\mathbf{x})|\varepsilon_f$ is still that high. The shapes of the two functions related to the temperature are depicted in Fig. 7.7.

We make the MATLAB program "`nm717.m`", which uses the routine "`sim_anl()`" to minimize a function

$$f(x) = x_1^4 - 16x_1^2 - 5x_1 + x_2^4 - 16x_2^2 - 5x_2 \tag{7.1.25}$$

and tries other routines such as "`opt_Nelder()`", "`fminsearch()`", and "`fminunc()`" for cross-checking. The results of running the program are summarized in Table 7.1, which shows that the routine "`sim_anl()`" may give us the global minimum even when some other routines fail to find it. But, even this routine based on the idea of simulated annealing cannot always succeed and its success/failure depends partially on the initial guess and partially on luck, while the success/failure of the other routines depends solely on the initial guess.

Table 7.1 Results of Running Several Optimization Routines with Various Initial Values

$\mathbf{x}_0$	`opt_Nelder()`	`fminsearch()`	`fminunc()`	`sim_anl()`
[0, 0]	[2.9035, 2.9035]	[2.9035, 2.9036]	[2.9036, 2.9036]	[2.8966, 2.9036]
	$(f^o = -156.66)$	$(f^o = -156.66)$	$(f^o = -156.66)$	$(f^o = -156.66)$
[−0.5,−1.0]	[2.9035, −2.7468]	[−2.7468, −2.7468]	[−2.7468, −2.7468]	[2.9029, 2.9028]
	$(f^o = -128.39)$	$(f^o = -100.12)$	$(f^o = -100.12)$	$(f^o = -156.66)$

7.1.8 Genetic Algorithm [W-7]

Genetic algorithm (GA) is a directed random search technique that is modeled on the natural evolution/selection process toward the survival of the fittest. The genetic operators deal with the individuals in a population over several generations to improve their fitness gradually. Individuals standing for possible solutions are often compared to chromosomes and represented by strings of binary numbers. Like the simulated annealing method, GA is also expected to find the global minimum solution even in the case where the objective function has several extrema, including local maxima, saddle points as well as local minima.

A so-called hybrid genetic algorithm [P-2] consists of initialization, evaluation, reproduction (selection), crossover, and mutation as depicted in Fig. 7.8

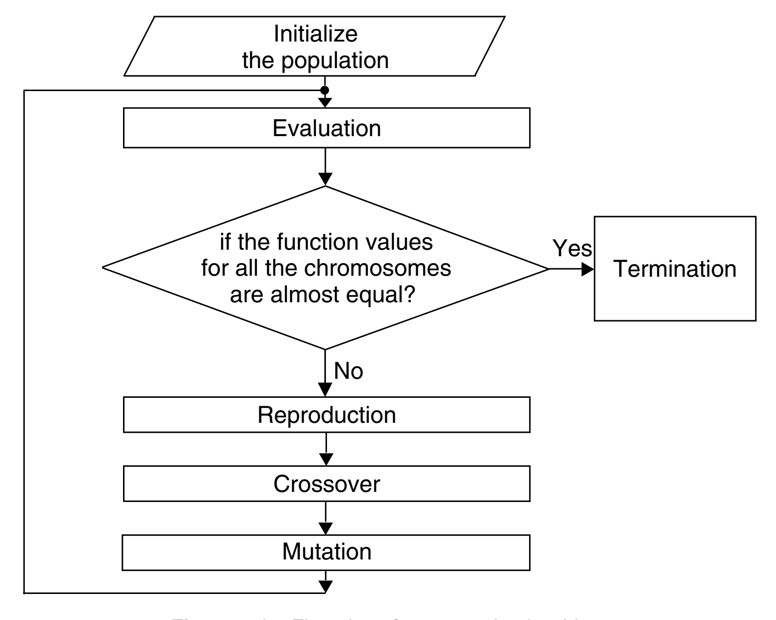


Figure 7.8 Flowchart for a genetic algorithm.

$N_p = 8,\ N = 2,\ N_b = [8\ 8]$

<table>
<tr><th colspan="2">pool P</th><th></th><th colspan="2">population X</th><th></th><th>fx</th></tr>
<tr><td>01100110</td><td>01100110</td><td></td><td>−1.0000</td><td>−1.0000</td><td></td><td>−10.00</td></tr>
<tr><td>01001111</td><td>10101011</td><td></td><td>−1.9020</td><td>−1.7059</td><td></td><td>−40.95</td></tr>
<tr><td>11110110</td><td>01101000</td><td></td><td>4.6471</td><td>−0.9216</td><td></td><td>44.67</td></tr>
<tr><td>01110111</td><td>11101111</td><td></td><td>−0.3333</td><td>4.3725</td><td></td><td>18.84</td></tr>
<tr><td>10101101</td><td>10110011</td><td>decode →</td><td>1.7843</td><td>2.0196</td><td>evaluate →</td><td>(−54.22)</td></tr>
<tr><td>11011011</td><td>11110110</td><td></td><td>3.5882</td><td>4.6471</td><td></td><td>19.71</td></tr>
<tr><td>11011000</td><td>00000001</td><td></td><td>3.4706</td><td>−4.9608</td><td></td><td>85.84</td></tr>
<tr><td>10011100</td><td>00011110</td><td></td><td>1.1176</td><td>−3.8235</td><td></td><td>−12.54</td></tr>
<tr><th colspan="2">random pairing</th><th></th><th colspan="2">↓ reproduction</th><th></th><th></th></tr>
<tr><td>a1 01111100</td><td>01111110 a2</td><td></td><td>−0.1209</td><td>−0.0466</td><td></td><td>0.28</td></tr>
<tr><td>b1 010|10111</td><td>1010101|1 b2</td><td></td><td>−1.5527</td><td>1.7356</td><td></td><td>−36.40</td></tr>
<tr><td>c1 1100|0010</td><td>100111|00 c2</td><td></td><td>2.6259</td><td>1.1550</td><td></td><td>−50.62</td></tr>
<tr><td>d1 10010011</td><td>11001111 d2</td><td></td><td>0.7713</td><td>3.1452</td><td></td><td>−44.58</td></tr>
<tr><td>e1 10101101</td><td>10110011 e2</td><td>← encode</td><td>1.7843</td><td>2.0196</td><td></td><td>−54.22</td></tr>
<tr><td>f1 110|00010</td><td>1101001|0 f2</td><td></td><td>2.6360</td><td>3.2601</td><td></td><td>(−74.73)</td></tr>
<tr><td>g1 1010|1101</td><td>101100|11 g2</td><td></td><td>1.7843</td><td>2.0196</td><td></td><td>−54.22</td></tr>
<tr><td>h1 10100001</td><td>01001010 h2</td><td></td><td>1.3160</td><td>−2.0846</td><td></td><td>−35.76</td></tr>
<tr><th colspan="2">crossover/mutuation ↓</th><th></th><th colspan="2"></th><th></th><th></th></tr>
<tr><td>a1 01111100</td><td>01111110 a2</td><td></td><td>−0.1373</td><td>−0.0588</td><td></td><td>0.31</td></tr>
<tr><td>b1' 01000010</td><td>10101010 b2'</td><td></td><td>−2.4118</td><td>1.6667</td><td></td><td>−46.12</td></tr>
<tr><td>c1' 11001101</td><td>100111|11 c2'</td><td></td><td>3.0392</td><td>1.2353</td><td></td><td>−52.96</td></tr>
<tr><td>d1 10010011</td><td>11001111 d2</td><td></td><td>0.7647</td><td>3.1176</td><td></td><td>−44.73</td></tr>
<tr><td>e1 10101101</td><td>10110011 e2</td><td>decode →</td><td>1.7843</td><td>2.0196</td><td></td><td>−54.22</td></tr>
<tr><td>f1' 11010111</td><td>11010011 f2'</td><td></td><td>3.4314</td><td>3.2745</td><td></td><td>(−69.94)</td></tr>
<tr><td>g1' 10100010</td><td>10110000 g2'</td><td></td><td>1.3529</td><td>1.9020</td><td></td><td>−43.50</td></tr>
<tr><td>h1 10100001</td><td>01001010 h2</td><td></td><td>1.3137</td><td>−2.0980</td><td></td><td>−35.88</td></tr>
</table>

. . . .

Figure 7.9 Reproduction/crossover mutation in one iteration of genetic algorithm.

and is summarized in the box below. The reproduction/crossover process is illustrated in Fig 7.9. This algorithm is cast into the routine "`genetic()`" and we append the following statements to the MATLAB program "`nm717.m`" in order to apply the routine for minimizing the function defined by Eq. (7.1.25). Interested readers are welcome to run the program with these statements appended and compare the result with those of using other routines. Note that like the simulated annealing, the routine based on the idea of GA cannot always succeed and its success/failure depends partially on the initial guess and partially on luck.

```
Np = 30; %population size
Nb = [12 12]; %the numbers of bits for representing each variable
Pc = 0.5; Pm = 0.01; %Probability of crossover/mutation
eta = 1; kmax = 100; %learning rate and the maximum # of iterations
[xo_gen,fo_gen] = genetic(f,x0,l,u,Np,Nb,Pc,Pm,eta,kmax)
```

HYBRID GENETIC ALGORITHM

Step 0. Pick the initial guess $\mathbf{x}_0 = [x_{01} \dots x_{0N}]$($N$: the dimension of the variable), the lower bound $\mathbf{l} = [l_1 \dots l_N]$, the upper bound $\mathbf{u} = [u_1 \dots u_N]$, the population size N_p, the vector $\mathbf{N}_b = [N_{b1} \dots N_{bN}]$ consisting of the numbers of bits assigned for the representation of each variable x_i, the probability of crossover P_c, the probability of mutation P_m, the learning rate $\eta(0 < \eta \le 1$, to be made small/large for slow/fast learning), and the maximum number of iterations $k_{\max} > 0$. Note that the dimensions of $\mathbf{x}_0$, $\mathbf{u}$, and $\mathbf{l}$ are all the same as N, which is the dimension of the variable $\mathbf{x}$ to be found and the population size N_p can not be greater than 2^{N_b} in order to avoid duplicated chromosomes and should be an even integer for constituting the mating pool in the crossover stage.

Step 1. Random Generation of Initial Population

Set $\mathbf{x}^o = \mathbf{x}_0$, $f^o = f(\mathbf{x}^o)$ and construct in a random way the initial population array X_1 that consists of N_p states (in the admissible region bounded by $\mathbf{u}$ and $\mathbf{l}$) including the initial state $\mathbf{x}_0$, by setting

$$X_1(1) = \mathbf{x}_0 \text{ and } X_1(k) = \mathbf{l} + \mathbf{rand}.^*(\mathbf{u} - \mathbf{l}) \quad \text{for } k = 2 : Np \qquad (7.1.26)$$

where **rand** is a random vector of the same dimension N as $\mathbf{x}_0$, $\mathbf{u}$, and $\mathbf{l}$. Then, encode each number of this population array into a binary string by

$$\begin{aligned} P_1(n, 1 + \textstyle\sum_{i=1}^{m-1} N_{bi} : \sum_{i=1}^{m} N_{bi}) \\ &= \text{binary representation of } X_1(n, m) \text{ with } N_{bm} \text{ bits} \\ &= (2^{N_{bm}} - 1)\frac{X_1(n, m) - l(m)}{u(m) - l(m)} \\ &\quad \text{for } n = 1 : N_p \text{ and } m = 1 : N \end{aligned} \qquad (7.1.27)$$

so that the whole population array becomes a pool array, each row of which is a chromosome represented by a binary string of $\sum_{i=1}^{N} N_{bi}$ bits.

Step 2. For $k = 1$ to $k_{\max}$, do the following:

1. Decode each number in the pool into a (decimal) number by

$$X_k(n, m) = \text{decimal representation of}$$
$$P_k\left(n, 1 + \sum_{i=1}^{m-1} N_{bi} : \sum_{i=1}^{m} N_{bi}\right) \text{ with } N_{bm} \text{ bits}$$
$$= P_k(n, \cdot)\frac{u(m) - l(m)}{2^{N_{bm}} - 1} + l(m)$$
$$\text{for } n = 1 : N_p \text{ and } m = 1 : N \qquad (7.1.28)$$

   and evaluate the value $f(n)$ of function for every row $X_k(n, :) = \mathbf{x}(n)$ corresponding to each chromosome and find the minimum $f_{\min} = f(n_b)$ corresponding to $X_k(n_b, :) = \mathbf{x}(n_b)$.
2. If $f_{\min} = f(n_b) < f^o$, then set $f^o = f(n_b)$ and $\mathbf{x}^o = \mathbf{x}(n_b)$.
3. Convert the function values into the values of fitness by

$$f_1(n) = \text{Max}_{n=1}^{N_p}\{f(n)\} - f(n) \qquad (7.1.29)$$

   which is nonnegative $\forall\ n = 1 : N_p$ and is large for a good chromosome.
4. If $\text{Max}_{n=1}^{N_p}\{f_1(n)\} \approx 0$, then terminate this procedure, declaring $\mathbf{x}^o$ as the best.
   Otherwise, in order to make more chromosomes around the best point $\mathbf{x}(n_b)$ in the next generation, use the reproduction rule

$$\mathbf{x}(n) \leftarrow \mathbf{x}(n) + \eta\frac{f_1(n_b) - f_1(n)}{f_1(n_b)}(\mathbf{x}(n_b) - \mathbf{x}(n)) \qquad (7.1.30)$$

   to get a new population X_{k+1} with $X_{k+1}(n, :) = \mathbf{x}(n)$ and encode it to reconstruct a new pool array P_{k+1} by Eq. (7.1.27).
5. Shuffle the row indices of the pool array for random mating of the chromosomes.
6. With the crossover probability P_c, exchange the tail part starting from some random bit of the numbers in two randomly paired chromosomes (rows of P_{k+1}) with each other's to get a new pool array P'_{k+1}.
7. With the mutation probability P_m, reverse a random bit of each number represented by chromosomes (rows of P'_{k+1}) to make a new pool array P_{k+1}.

```
function [xo,fo] = genetic(f,x0,l,u,Np,Nb,Pc,Pm,eta,kmax)
% Genetic Algorithm to minimize f(x) s.t. l <= x <= u
N = length(x0);
if nargin < 10, kmax = 100; end %# of iterations(generations)
if nargin < 9|eta > 1|eta <= 0, eta = 1; end %learning rate(0 < eta < 1)
if nargin < 8, Pm = 0.01; end %probability of mutation
if nargin < 7, Pc = 0.5; end %probability of crossover
if nargin < 6, Nb = 8*ones(1,N); end %# of genes(bits) for each variable
if nargin < 5, Np = 10; end %population size(number of chromosomes)
%Initialize the population pool
NNb = sum(Nb);
xo = x0(:)'; l = l(:)'; u = u(:)';
fo = feval(f,xo);
X(1,:) = xo;
for n = 2:Np, X(n,:) = l + rand(size(x0)).*(u - l);  end %Eq.(7.1.26)
P = gen_encode(X,Nb,l,u); %Eq.(7.1.27)
for k = 1:kmax
  X = gen_decode(P,Nb,l,u); %Eq.(7.1.28)
  for n = 1:Np, fX(n) = feval(f,X(n,:)); end
  [fxb,nb] = min(fX); %Selection of the fittest
  if fxb < fo, fo = fxb; xo = X(nb,:); end
  fX1 = max(fxs) - fX; %make the nonnegative fitness vector by Eq.(7.1.29)
  fXm = fX1(nb);
  if fXm < eps, return; end %terminate if all the chromosomes are equal
  %Reproduction of next generation
  for n = 1:Np
     X(n,:) = X(n,:) + eta*(fXm - fX1(n))/fXm*(X(nb,:) - X(n,:)); %Eq.(7.1.30)
  end
  P = gen_encode(X,Nb,l,u);
  %Mating/Crossover
  is = shuffle([1:Np]);
  for n = 1:2:Np - 1
    if rand < Pc, P(is(n:n + 1),:) = crossover(P(is(n:n + 1),:),Nb); end
  end
  %Mutation
  P = mutation(P,Nb,Pm);
end
```

```
function P = gen_encode(X,Nb,l,u)
% encode a population(X) of state into an array(P) of binary strings
Np=size(X,1); %population size
N = length(Nb); %dimension of the variable(state)
for n = 1:Np
   b2 = 0;
   for m = 1:N
      b1 = b2+1; b2 = b2 + Nb(m);
      Xnm =(2^Nb(m)- 1)*(X(n,m) - l(m))/(u(m) - l(m)); %Eq.(7.1.27)
      P(n,b1:b2) = dec2bin(Xnm,Nb(m)); %encoding to binary strings
   end
end
```

```
function X = gen_decode(P,Nb,l,u)
% decode an array of binary strings(P) into a population(X) of state
Np = size(P,1); %population size
N = length(Nb); %dimension of the variable(state)
for n = 1:Np
   b2 = 0;
   for m = 1:N
      b1 = b2 + 1; b2 = b1 + Nb(m) - 1; %Eq.(7.1.28)
      X(n,m) = bin2dec(P(n,b1:b2))*(u(m) - l(m))/(2^Nb(m) - 1) + l(m);
   end
end
```

```
function chrms2 = crossover(chrms2,Nb)
% crossover between two chromosomes
Nbb = length(Nb);
b2 = 0;
for m = 1:Nbb
   b1 = b2 + 1; bi = b1 + mod(floor(rand*Nb(m)),Nb(m)); b2 = b2 + Nb(m);
   tmp = chrms2(1,bi:b2);
   chrms2(1,bi:b2) = chrms2(2,bi:b2);
   chrms2(2,bi:b2) = tmp;
end
```

```
function P = mutation(P,Nb,Pm) % mutation
Nbb = length(Nb);
for n = 1:size(P,1)
   b2 = 0;
   for m = 1:Nbb
      if rand < Pm
        b1 = b2 + 1; bi = b1 + mod(floor(rand*Nb(m)),Nb(m)); b2 = b2 + Nb(m);
        P(n,bi) = ~P(n,bi);
      end
   end
end
```

```
function is = shuffle(is) % shuffle
N = length(is);
for n = N:-1:2
   in = ceil(rand*(n - 1)); tmp = is(in);
   is(in) = is(n); is(n) = tmp; %swap the n-th element with the in-th one
end
```

7.2 CONSTRAINED OPTIMIZATION [L-2, CHAPTER 10]

In this section, only the concept of constrained optimization is introduced. The explanation for the usage of the corresponding MATLAB routines is postponed until the next section.

7.2.1 Lagrange Multiplier Method

A class of common optimization problems subject to equality constraints may be nicely handled by the Lagrange multiplier method. Consider an optimization problem with M equality constraints.

$$\text{Min } f(\mathbf{x}) \tag{7.2.1a}$$

$$\text{s.t.} \quad \mathbf{h}(\mathbf{x}) = \begin{bmatrix} h_1(\mathbf{x}) \\ h_2(\mathbf{x}) \\ : \\ h_M(\mathbf{x}) \end{bmatrix} = \mathbf{0} \tag{7.2.1b}$$

According to the Lagrange multiplier method, this problem can be converted to the following unconstrained optimization problem:

$$\text{Min } l(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \boldsymbol{\lambda}^T \mathbf{h}(\mathbf{x}) = f(\mathbf{x}) + \sum_{m=1}^{M} \lambda_m h_m(\mathbf{x}) \tag{7.2.2}$$

The solution of this problem, if it exists, can be obtained by setting the derivatives of this new objective function $l(\mathbf{x}, \boldsymbol{\lambda})$ with respect to $\mathbf{x}$ and $\boldsymbol{\lambda}$ to zero:

$$\frac{\partial}{\partial \mathbf{x}} l(\mathbf{x}, \boldsymbol{\lambda}) = \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}) + \boldsymbol{\lambda}^T \frac{\partial}{\partial \mathbf{x}} \mathbf{h}(\mathbf{x}) = \nabla f(\mathbf{x}) + \sum_{m=1}^{M} \lambda_m \nabla h_m(\mathbf{x}) = \mathbf{0} \quad (7.2.3a)$$

$$\frac{\partial}{\partial \boldsymbol{\lambda}} l(\mathbf{x}, \boldsymbol{\lambda}) = \mathbf{h}(\mathbf{x}) = \mathbf{0} \quad (7.2.3b)$$

Note that the solutions for this system of equations are the extrema of the objective function. We may know if they are minima/maxima, from the positive/negative-definiteness of the second derivative (Hessian matrix) of $l(\mathbf{x}, \boldsymbol{\lambda})$ with respect to $\mathbf{x}$. Let us see the following examples.

Remark 7.2. Inequality Constraints with the Lagrange Multiplier Method.

Even though the optimization problem involves inequality constraints like $g_j(\mathbf{x}) \le 0$, we can convert them to equality constraints by introducing the (nonnegative) slack variables y_j^2 as

$$g_j(\mathbf{x}) + y_j^2 = 0 \quad (7.2.4)$$

Then, we can use the Lagrange multiplier method to handle it like an equality-constrained problem.

Example 7.1. Minimization by the Lagrange Multiplier Method.

Consider the following minimization problem subject to a single equality constraint:

$$\text{Min } f(\mathbf{x}) = x_1^2 + x_2^2 \quad (E7.1.1a)$$

$$\text{s.t.} \quad h(\mathbf{x}) = x_1 + x_2 - 2 = 0 \quad (E7.1.1b)$$

We can substitute the equality constraint $x_2 = 2 - x_1$ into the objective function (E7.1.1a) so that this problem becomes an unconstrained optimization problem as

$$\text{Min } f(x_1) = x_1^2 + (2 - x_1)^2 = 2x_1^2 - 4x_1 + 4 \quad (E7.1.2)$$

which can be easily solved by setting the derivative of this new objective function with respect to x_1 to zero.

$$\frac{\partial}{\partial x_1} f(x_1) = 4x_1 - 4 = 0, \qquad x_1 = 1, \; x_2 \overset{(E7.1.1b)}{=} 2 - x_1 = 1 \quad (E7.1.3)$$

Alternatively, we can apply the Lagrange multiplier method as follows:

$$\text{Min } l(\mathbf{x}, \lambda) \overset{(7.2.2)}{=} x_1^2 + x_2^2 + \lambda(x_1 + x_2 - 2) \quad (E7.1.4)$$

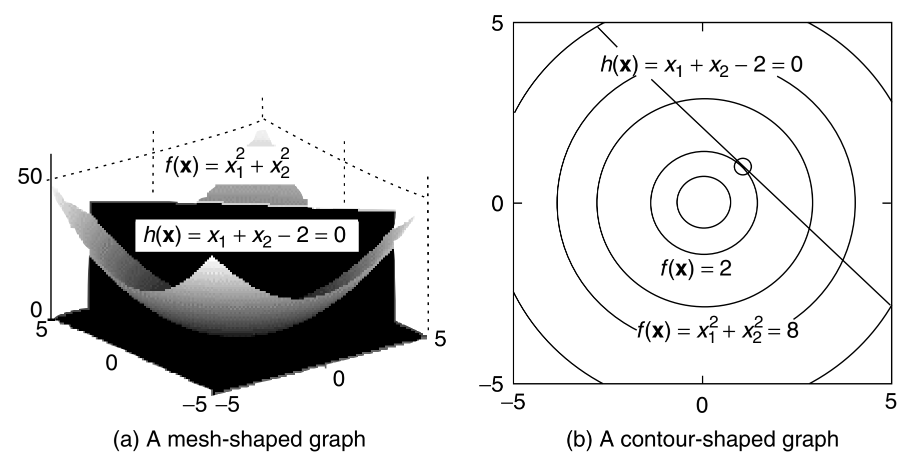


Figure 7.10 The objective function with constraint for Example 7.1.

$$\frac{\partial}{\partial x_1} l(\mathbf{x}, \lambda) \overset{(7.2.3a)}{=} 2x_1 + \lambda = 0, \qquad x_1 = -\lambda/2 \tag{E7.1.5a}$$

$$\frac{\partial}{\partial x_2} l(\mathbf{x}, \lambda) \overset{(7.2.3a)}{=} 2x_2 + \lambda = 0, \qquad x_2 = -\lambda/2 \tag{E7.1.5b}$$

$$\frac{\partial}{\partial \lambda} l(\mathbf{x}, \lambda) \overset{(7.2.3b)}{=} x_1 + x_2 - 2 = 0 \tag{E7.1.5c}$$

$$x_1 + x_2 \overset{(E7.1.5c)}{=} 2 \overset{(E7.1.5a,b)}{\rightarrow} -\lambda/2 - \lambda/2 = -\lambda = 2, \qquad \lambda = -2 \tag{E7.1.6}$$

$$x_1 \overset{(E7.1.5a)}{=} -\lambda/2 = 1, \quad x_2 \overset{(E7.1.5b)}{=} -\lambda/2 = 1 \text{ (Fig. 7.10)} \tag{E7.1.7}$$

In this example, the substitution of (linear) equality constraints is more convenient than the Lagrange multiplier method. However, it is not always the case, as illustrated by the next example.

Example 7.2. Minimization by the Lagrange Multiplier Method.

Consider the following minimization problem subject to a single nonlinear equality constraint:

$$\text{Min } f(\mathbf{x}) = x_1 + x_2 \tag{E7.2.1a}$$

$$\text{s.t.} \quad h(\mathbf{x}) = x_1^2 + x_2^2 - 2 = 0 \tag{E7.2.1b}$$

Noting that it is absurd to substitute the equality constraint (E7.2.1b) into the objective function (E7.2.1a), we apply the Lagrange multiplier method as below.

$$\text{Min } l(\mathbf{x}, \lambda) \overset{(7.2.2)}{=} x_1 + x_2 + \lambda(x_1^2 + x_2^2) \tag{E7.2.3}$$

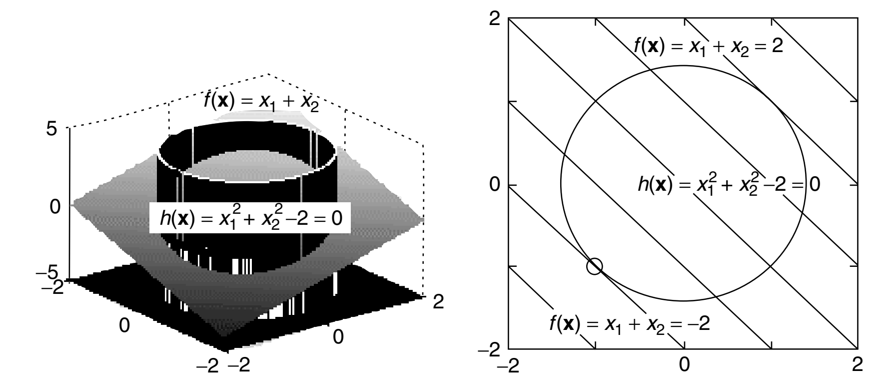


Figure 7.11 The objective function with constraint for Example 7.2.

$$\frac{\partial}{\partial x_1} l(\mathbf{x}, \lambda) \overset{(7.2.3a)}{=} 1 + 2\lambda x_1 = 0, \qquad x_1 = -1/2\lambda \tag{E7.2.4a}$$

$$\frac{\partial}{\partial x_2} l(\mathbf{x}, \lambda) \overset{(7.2.3a)}{=} 1 + 2\lambda x_2 = 0, \qquad x_2 = -1/2\lambda \tag{E7.2.4b}$$

$$\frac{\partial}{\partial \lambda} l(\mathbf{x}, \lambda) \overset{(7.2.3b)}{=} x_1^2 + x_2^2 - 2 = 0 \tag{E7.2.4c}$$

$$x_1^2 + x_2^2 \overset{(E7.2.4c)}{=} 2 \overset{(E7.2.4a,b)}{\rightarrow} (-1/2\lambda)^2 + (-1/2\lambda)^2 = 2, \quad \lambda = \pm 1/2 \tag{E7.2.5}$$

$$x_1 \overset{(E7.2.4a)}{=} -1/2\lambda = \mp 1, \qquad x_2 \overset{(E7.2.4b)}{=} -1/2\lambda = \mp 1 \tag{E7.2.6}$$

Now, in order to tell whether each of these is a minimum or a maximum, we should determine the positive/negative-definiteness of the second derivative (Hessian matrix) of $l(\mathbf{x}, \boldsymbol{\lambda})$ with respect to $\mathbf{x}$.

$$H = \frac{\partial^2}{\partial \mathbf{x}^2} l(\mathbf{x}, \boldsymbol{\lambda}) = \begin{bmatrix} \partial^2 l/\partial x_1^2 & \partial^2 l/\partial x_1 \partial x_2 \\ \partial^2 l/\partial x_2 \partial x_1 & \partial^2 l/\partial x_2^2 \end{bmatrix} = \begin{bmatrix} 2\lambda & 0 \\ 0 & 2\lambda \end{bmatrix} \tag{E7.2.7}$$

This matrix is positive/negative-definite if the sign of λ is positive/negative. Therefore, the solution $(x_1, x_2) = (-1, -1)$ corresponding to $\lambda = 1/2$ is a (local) minimum that we want to get, while the solution $(x_1, x_2) = (1, 1)$ corresponding to $\lambda = -1/2$ is a (local) maximum (see Fig. 7.11).

7.2.2 Penalty Function Method

This method is practically very useful for dealing with the general constrained optimization problems involving equality/inequality constraints. It is really

attractive for optimization problems with fuzzy or loose constraints that are not so strict with zero tolerance.

Consider the following problem.

$$\text{Min } f(\mathbf{x}) \tag{7.2.5a}$$

$$\text{s.t.} \quad \mathbf{h}(\mathbf{x}) = \begin{bmatrix} h_1(\mathbf{x}) \\ : \\ h_M(\mathbf{x}) \end{bmatrix} = \mathbf{0}, \qquad \mathbf{g}(\mathbf{x}) = \begin{bmatrix} g_1(\mathbf{x}) \\ : \\ g_L(\mathbf{x}) \end{bmatrix} \le \mathbf{0} \tag{7.2.5b}$$

The penalty function method consists of two steps. The first step is to construct a new objective function

$$\text{Min } l(\mathbf{x}) = f(\mathbf{x}) + \sum_{m=1}^{M} w_m h_m^2(\mathbf{x}) + \sum_{m=1}^{L} v_m \psi(g_m(\mathbf{x})) \tag{7.2.6}$$

by including the constraint terms in such a way that violating the constraints would be penalized through the large value of the constraint terms in the objective function, while satisfying the constraints would not affect the objective function. The second step is to minimize the new objective function with no constraints by using the method that is applicable to unconstrained optimization problems, but a non-gradient-based approach like the Nelder method. Why don't we use a gradient-based optimization method? Because the inequality constraint terms $v_m\psi_m(g_m(\mathbf{x}))$ attached to the objective function are often determined to be zero as long as $\mathbf{x}$ stays inside the (permissible) region satisfying the corresponding constraint ($g_m(\mathbf{x}) \le 0$) and to increase very steeply (like $\psi_m(g_m(\mathbf{x})) = \exp(e_m g_m(\mathbf{x}))$ as $\mathbf{x}$ goes out of the region; consequently, the gradient of the new objective function may not carry useful information about the direction along which the value of the objective function decreases.

From an application point of view, it might be a good feature of this method that we can make the weighting coefficient (w_m,v_m, and e_m) on each penalizing constraint term either large or small depending on how strictly it should be satisfied.

Let us see the following example.

Example 7.3. Minimization by the Penalty Function Method.

Consider the following minimization problem subject to several nonlinear inequality constraints:

$$\text{Min } f(\mathbf{x}) = \{(x_1 + 1.5)^2 + 5(x_2 - 1.7)^2\}\{(x_1 - 1.4)^2 + 0.6(x_2 - 0.5)^2\} \tag{E7.3.1a}$$

$$\text{s.t.} \quad \mathbf{g}(\mathbf{x}) = \begin{bmatrix} -x_1 \\ -x_2 \\ 3x_1 - x_1x_2 + 4x_2 - 7 \\ 2x_1 + x_2 - 3 \\ 3x_1 - 4x_2^2 - 4x_2 \end{bmatrix} \le \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{E7.3.1b}$$

According to the penalty function method, we construct a new objective function (7.2.6) as

$$\text{Min } l(\mathbf{x}) = \{(x_1 + 1.5)^2 + 5(x_2 - 1.7)^2\}\{(x_1 - 1.4)^2 + 0.6(x_2 - 0.5)^2\} + \sum_{m=1}^{5} v_m \psi_m(g_m(\mathbf{x})) \tag{E7.3.2a}$$

where

$$v_m = 1, \quad \psi_m(g_m(\mathbf{x})) = \begin{cases} 0 & \text{if } g_m(\mathbf{x}) \le 0 \text{ (constraint satisfied)} \\ \exp(e_m g_m(\mathbf{x})) & \text{if } g_m(\mathbf{x}) > 0 \text{ (constraint violated)} \end{cases},$$
$$e_m = 1 \ \forall\ m = 1, \ldots, 5 \tag{E7.3.2b}$$

```
%nm722 for Ex.7.3
% to solve a constrained optimization problem by penalty ftn method.
clear, clf
f ='f722p';
x0=[0.4 0.5]
TolX = 1e-4; TolFun = 1e-9; alpha0 = 1;
[xo_Nelder,fo_Nelder] = opt_Nelder(f,x0) %Nelder method
[fc_Nelder,fo_Nelder,co_Nelder] = f722p(xo_Nelder) %its results
[xo_s,fo_s] = fminsearch(f,x0) %MATLAB built-in fminsearch()
[fc_s,fo_s,co_s] = f722p(xo_s) %its results
  % including how the constraints are satisfied or violated
xo_steep = opt_steep(f,x0,TolX,TolFun,alpha0) %steepest descent method
[fc_steep,fo_steep,co_steep] = f722p(xo_steep) %its results
[xo_u,fo_u] = fminunc(f,x0); % MATLAB built-in fminunc()
[fc_u,fo_u,co_u] = f722p(xo_u) %its results
```

```
function [fc,f,c] = f722p(x)
f=((x(1)+ 1.5)^2 + 5*(x(2)- 1.7)^2)*((x(1)- 1.4)^2 + .6*(x(2)-.5)^2);
c=[-x(1); -x(2); 3*x(1) - x(1)*x(2) + 4*x(2) - 7;
     2*x(1)+ x(2) - 3; 3*x(1) - 4*x(2)^2 - 4*x(2)]; %constraint vector
v=[1 1 1 1 1]; e = [1 1 1 1 1]'; %weighting coefficient vector
fc = f +v*((c > 0).*exp(e.*c)); %new objective function
```

<pre>>> nm722
 	
xo_Nelder = 1.2118   0.5765
fo_Nelder = 0.5322 %min value
co_Nelder = -1.2118
            -0.5765
            -1.7573 %high margin
            -0.0000 %no margin
            -0.0000 %no margin
xo_s = 1.2118   0.5765
fo_s = 0.5322 %min value</pre>
	<pre>xo_steep = 1.2768 0.5989
	fo_steep = 0.2899 %not a minimum
	co_steep = -1.2768
	           -0.5989
	           -1.5386
	            0.1525 %violating
	           -0.0001
	Warning: .. Gradient must be provided
	 ....
	 Maximum # of function evaluations
	 exceeded;
	xo_u = 1.2843 0.6015
	fo_u = 0.2696 %not a minium</pre>

Note that the shape of the penalty function as well as the values of the weighting coefficients is set by the users to cope with their own problems. Then, we apply an unconstrained optimization technique like the Nelder–Mead method, which is not a gradient-based approach. Here, we make the program "`nm722.m`", which applies not only the routine "`opt_Nelder()`" and the MATLAB built-in routine "`fminsearch()`" for cross-check, but also the routine "`opt_steep()`" and the MATLAB built-in routine "`fminunc()`" in order to show that the gradient-based methods do not work well. To our expectation, the running results listed above and depicted in Fig. 7.12 show that, for the objective function (E7.3.2a) augmented with the penalized constraint terms, the gradient-based routines "`opt_steep()`" and "`fminunc()`" are not so effective as the non-gradient-based routines "`opt_Nelder()`" and "`fminsearch()`" in finding the constrained

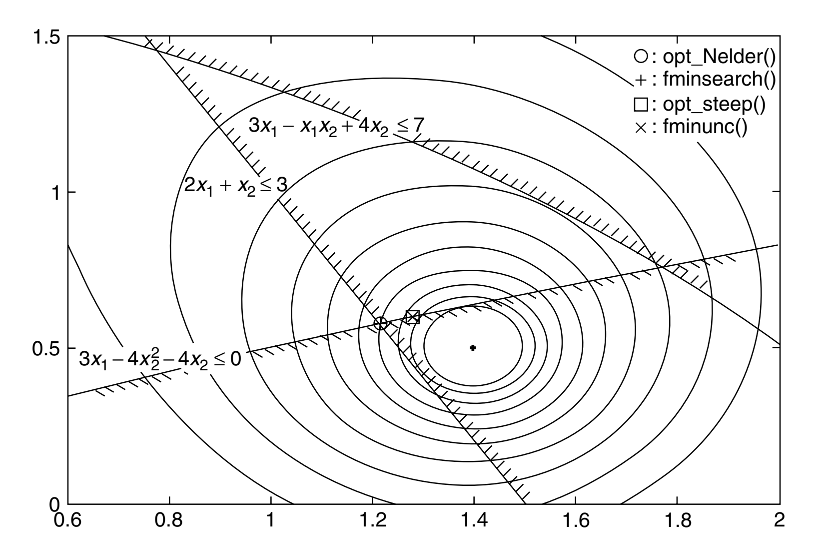


Figure 7.12 The contours for the objective function (E7.3.1a) and the admissible region satisfying the inequality constraints.

minimum, which is on the intersection of the two boundary curves corresponding to the fourth and fifth constraints of (E7.3.1b).

7.3 MATLAB BUILT-IN ROUTINES FOR OPTIMIZATION

In this section, we apply several MATLAB built-in unconstrained optimization routines including "`fminsearch()`" and "`fminunc()`" to the same problem, expecting that their nuances will be clarified. Our intention is not to compare or evaluate the performances of these sophisticated routines, but rather to give the readers some feelings for their functional differences. We also introduce the routine "`linprog()`" implementing Linear Programming (LP) scheme and "`fmincon()`" designed for attacking the (most challenging) constrained optimization problems. Interested readers are encouraged to run the tutorial routines "`optdemo`" or "`tutdemo`", which demonstrate the usages and performances of the representative built-in optimization routines such as "`fminunc()`" and "`fmincon()`".

7.3.1 Unconstrained Optimization

In order to try applying the unconstrained optimization routines introduced in Section 7.1 and see how they work, we made the MATLAB program "`nm731_1.m`", which uses those routines for solving the problem

$$\text{Min } f(\mathbf{x}) = (x_1 - 0.5)^2(x_1 + 1)^2 + (x_2 + 1)^2(x_2 - 1)^2 \tag{7.3.1}$$

where the contours and the (local) maximum/minimum/saddle points of this objective function are depicted in Fig. 7.13.

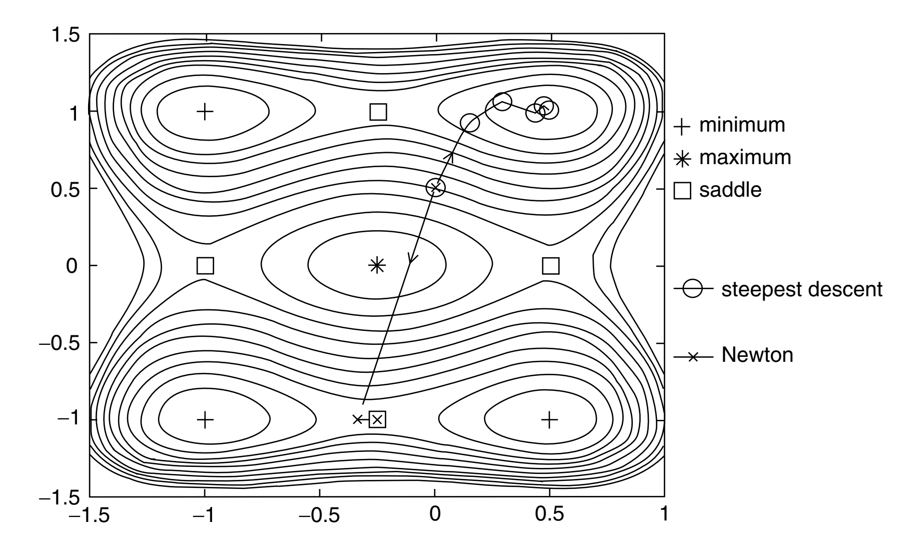


Figure 7.13 The contours, minima, maxima, and saddle points of the objective function (7.3.1).

```
%nm731_1
% to minimize an objective function f(x) by various methods.
clear, clf
% An objective function and its gradient function
f = inline('(x(1) - 0.5).^2.*(x(1) + 1).^2 + (x(2)+1).^2.*(x(2) - 1).^2','x');
g0 = '[2*(x(1)- 0.5)*(x(1)+ 1)*(2*x(1)+ 0.5) 4*(x(2)^2 - 1).*x(2)]';
g = inline(g0,'x');
x0 = [0 0.5] %initial guess
[xon,fon] = opt_Nelder(f,x0) %min point, its ftn value by opt_Nelder
[xos,fos] = fminsearch(f,x0) %min point, its ftn value by fminsearch()
[xost,fost] = opt_steep(f,x0) %min point, its ftn value by opt_steep()
TolX = 1e-4; MaxIter = 100;
xont = Newtons(g,x0,TolX,MaxIter);
xont,f(xont) %minimum point and its function value by Newtons()
[xocg,focg] = opt_conjg(f,x0) %min point, its ftn value by opt_conjg()
[xou,fou] = fminunc(f,x0) %min point, its ftn value by fminunc()
```

Noting that it depends mainly on the initial value $\mathbf{x}_0$ whether each routine succeeds in finding a minimum point, we summarize the results of running those routines with various initial values in Table 7.2. It can be seen from this table that the gradient-based optimization routines like "`opt_steep()`", "`Newtons()`", "`opt_conj()`", and "`fminunc()`" sometimes get to a saddle point or even a maximum point (Remark 7.1) and that the routines do not always approach the extremum that is closest to the initial point. It is interesting to note that even the non-gradient-based MATLAB built-in routine "`fminsearch()`" may get lost, while our routine "`opt_Nelder()`" works well for this case. We cannot, however, conclude that this routine is better than that one based on only one trial, because there may be some problems for which the MATLAB built-in routine works well, but our routine does not. What we can state over this happening is that no human work is free from defect.

Now, we will see a MATLAB built-in routine "`lsqnonlin(f,x0,l,u, options,p1,..)`", which presents a nonlinear least-squares (NLLS) solution to

Table 7.2 Results of Running Several Unconstrained Optimization Routines with Various Initial Values

$\mathbf{x}_0$	opt_Nelder	fminsearch	opt_steep	Newtons	opt_conjg	fminunc
[0, 0]	[−1, 1] (minimum)	[0.5, 1] (minimum)	[0.5, 0] (saddle)	[−0.25, 0] (maximum)	[0.5, 0] (saddle)	[0.5, 0] (saddle)
[0, 0.5]	[0.5, 1] (minimum)	[0.02, 1] (lost)	[0.5, 1] (minimum)	[−0.25, −1] (saddle)	[0.5, 1] (minimum)	[0.5, 1] (minimum)
[0.4, 0.5]	[0.5, 1] (minimum)	[0.5, 1] (minimum)	[0.5, 1] (minimum)	[0.5, −1] (minimum)	[0.5, 1] (minimum)	[0.5, 1] (minimum)
[−0.5, 0.5]	[0.5, 1] (minimum)	[−1, 1] (minimum)	[−1, 1] (minimum)	[−0.25, −1] (saddle)	[−1, 1] (minimum)	[−1, 1] (minimum)
[−0.8, 0.5]	[−1, 1] (minimum)	[−1, 1] (minimum)	[−1, 1] (minimum)	[−1, −1] (minimum)	[−1, 1] (minimum)	[−1, 1] (minimum)

the minimization problem

$$\text{Min} \sum_{n=1}^{N} f_n^2(\mathbf{x}) \tag{7.3.2}$$

The routine needs at least the vector or matrix function $\mathbf{f}(\mathbf{x})$ and the initial guess $\mathbf{x}_0$ as its first and second input arguments, where the components of $\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}) \cdots f_N(\mathbf{x})]^T$ are squared, summed, and then minimized over $\mathbf{x}$. In order to learn the usage and function of this routine, we made the MATLAB program "`nm731_2.m`", which uses it to find a second-degree polynomial approximating the function

$$y = f(x) = \frac{1}{1 + 8x^2} \tag{7.3.3}$$

For verification, the result of using the NLLS routine "`lsqnonlin()`" is compared with that obtained from directly applying the routine "`polyfits()`" introduced in Section 3.8.2.

```
>> nm731_2
   ao_lsq = [-0.1631  -0.0000  0.4653], ao_fit = [-0.1631  -0.0000  0.4653]
```

```
%nm731_2 try using lsqnonlin() for a vector-valued objective ftn F(x)
clear, clf
N = 3; a0 = zeros(1,N); %the initial guess of polynomial coefficient vector
ao_lsq = lsqnonlin('f731_2',a0) %parameter estimate by lsqnonlin()
xx = -2+[0:400]/50;  fx = 1./(1+8*xx.*xx);
ao_fit = polyfits(xx,fx,N - 1) %parameter estimate by polyfits()
```

```
function F = f731_2(a)
%error between the polynomial a(x) and f(x) = 1/(1+8x^2)
xx = -2 +[0:200]/50; F = polyval(a,xx) - 1./(1+8*xx.*xx);
```

7.3.2 Constrained Optimization

Generally, constrained optimization is very complicated and difficult to deal with. So we will not cover the topic in details here and instead, will just introduce the powerful MATLAB built-in routine "`fmincon()`", which makes us relieved from a big headache.

This routine is well-designed for attacking the optimization problems subject to some constraints:

```
function [c,ceq] = f722c(x)
c = [-x(1); -x(2); 3*x(1) - x(1)*x(2) + 4*x(2)- 7;
      2*x(1)+ x(2)- 3; 3*x(1)- 4*x(2)^2 - 4*x(2)]; %inequality constraints
ceq = []; %equality constraints
```

⟨Usage of the MATLAB 6.x built-in function "`fmincon()`"⟩

```
[xo,fo,.] = fmincon('ftn',x0,A,b,Aeq,beq,l,u,'nlcon',options,p1,p2,.)
```

- *Input Arguments* (at least four input arguments `'ftn'`,`x0`,`A` and `b` required)

`'ftn'` : an objective function $f(\mathbf{x})$ to be minimized, usually defined in an M-file, but can be defined as an inline function, which will remove the necessity of quotes(`''`).
`x0` : an initial guess $\mathbf{x}_0$ of the solution
`A,b` : a linear inequality constraints $A\mathbf{x} \le \mathbf{b}$; to be given as `[]` if not applied.
`Aeq,beq`: a linear equality constraints $A_{eq}\mathbf{x} = \mathbf{b}_{eq}$; to be given as `[]` if not applied.
`l,u` : lower/upper bound vectors such that $\mathbf{l} \le \mathbf{x} \le \mathbf{u}$; to be given as `[]` if not applied, set `l(i) = -inf/u(i) = inf if x(i)` is not bounded below/above.
`'nlcon'`: a nonlinear constraint function defined in an M-file, supposed to return the two output arguments for a given $\mathbf{x}$; the first one being the LHS (vector) of inequality constraints $\mathbf{c}(\mathbf{x}) \le \mathbf{0}$ and the second one being the LHS (vector) of equality constraints $\mathbf{c}_{eq}(\mathbf{x}) = \mathbf{0}$; to be given as `[]` if not applied.
`options`: used for setting the display parameter, the tolerances for $\mathbf{x}_o$ and $f(\mathbf{x}_o)$, and so on; to be given as `[]` if not applied. For details, type '`help optimset`' into the MATLAB command window.
`p1,p2,.`: the problem-dependent parameters to be passed to the objective function $f(\mathbf{x})$ and the nonlinear constraint functions $\mathbf{c}(\mathbf{x})$, $\mathbf{c}_{eq}(\mathbf{x})$.

- *Output Arguments*

`xo` : the minimum point ($\mathbf{x}_o$) reached in the permissible region satisfying the constraints
`fo` : the minimized function value $f(\mathbf{x}_o)$

```
%nm732_1 to solve a constrained optimization problem by fmincon()
clear, clf
ftn='((x(1) + 1.5)^2 + 5*(x(2) - 1.7)^2)*((x(1)-1.4)^2 + .6*(x(2)-.5)^2)';
f722o = inline(ftn,'x');
x0 = [0 0.5] %initial guess
A = []; B = []; Aeq = []; Beq = []; %no linear constraints
l = -inf*ones(size(x0)); u = inf*ones(size(x0)); % no lower/upperbound
options = optimset('LargeScale','off'); %just [] is OK.
[xo_con,fo_con] = fmincon(f722o,x0,A,B,Aeq,Beq,l,u,'f722c',options)
[co,ceqo] = f722c(xo_con) % to see how constraints are.
```

$$\text{Min } f(\mathbf{x}) \tag{7.3.4}$$

$$\text{s.t. } A\mathbf{x} \le \mathbf{b},\ \ A_{eq}\mathbf{x} = \mathbf{b}_{eq},\ \ \mathbf{c}(\mathbf{x}) \le \mathbf{0},\ \ \mathbf{c}_{eq}(\mathbf{x}) = \mathbf{0} \text{ and } \mathbf{l} \le \mathbf{x} \le \mathbf{u} \tag{7.3.5}$$

A part of its usage can be seen by typing '`help fmincon`' into the MATLAB command window as summarized in the above box. We make the MATLAB program "`nm732_1.m`", which uses the routine "`fmincon()`" to solve the problem presented in Example 7.3. Interested readers are welcomed to run it and observe the result to check if it agrees with that of Example 7.3.

There are two more MATLAB built-in routines to be introduced in this section. One is

```
"fminimax('ftn',x0,A,b,Aeq,beq,l,u,'nlcon',options,p1,..)",
```

which is focused on minimizing the maximum among several components of the vector/matrix-valued objective function $\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}) \cdots f_N(\mathbf{x})]^T$ subject to some constraints as described below. Its usage is almost the same as that of "`fmincon()`".

$$\underset{\mathbf{x}}{\text{Min}}\{\underset{n}{\text{Max}}\{f_n(\mathbf{x})\}\} \tag{7.3.6}$$

$$\text{s.t. } A\mathbf{x} \le \mathbf{b},\ \ A_{eq}\mathbf{x} = \mathbf{b}_{eq},\ \ \mathbf{c}(\mathbf{x}) \le \mathbf{0},\ \ \mathbf{c}_{eq}(\mathbf{x}) = \mathbf{0},\ \text{ and } \mathbf{l} \le \mathbf{x} \le \mathbf{u} \tag{7.3.7}$$

The other is the constrained linear least-squares (LLS) routine

```
"lsqlin(C,d,A,b,Aeq,beq,l,u,x0,options,p1,..)",
```

whose job is to solve the problem

$$\underset{\mathbf{x}}{\text{Min}}\ ||C\mathbf{x} - \mathbf{d}||^2 \tag{7.3.8}$$

$$\text{s.t.} \quad A\mathbf{x} \le \mathbf{b}, \quad A_{eq}\mathbf{x} = \mathbf{b}_{eq} \quad \text{and} \quad \mathbf{l} \le \mathbf{x} \le \mathbf{u} \tag{7.3.9}$$

In order to learn the usage and function of this routine, we make the MATLAB program "`nm732_2.m`", which uses both "`fminimax()`" and "`lsqlin()`" to find a second-degree polynomial approximating the function (7.3.3) and compares the results with that of applying the routine "`lsqnonlin()`" introduced in the previous section for verification. From the plotting result depicted in Fig. 7.14, note the following.

- We attached no constraints to the "`fminimax()`" routine, so it yielded the approximate polynomial curve minimizing the maximum deviation from $f(x)$.
- We attached no constraints to the constrained linear least-squares routine "`lsqlin()`" either, so it yielded the approximate polynomial curve minimizing the sum (integral) of squared deviation from $f(x)$, which is

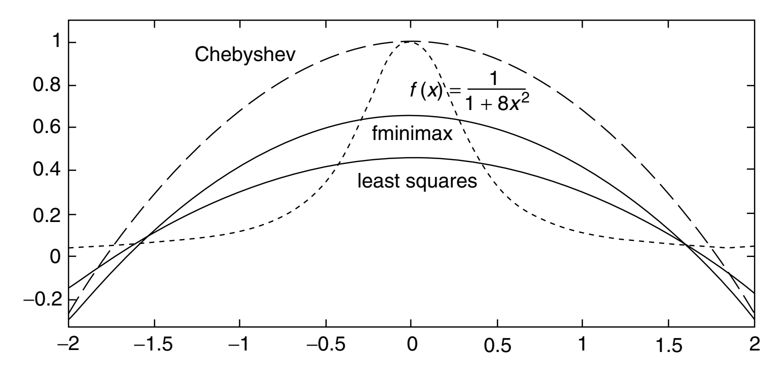


Figure 7.14 Approximation of a curve by a second-degree polynomial function based on the minimax, least-squares, and Chebyshev methods.

the same as the (unconstrained) least squares solution obtained by using the routine "`lsqnonlin()`".

- Another MATLAB built-in routine "`lsqnonneg()`" gives us a nonnegative LS (NLS) solution to the problem (7.3.8).

```
%nm732_2: uses fminimax() for a vector-valued objective ftn f(x)
clear, clf
f = inline('1./(1+8*x.*x)','x');
f73221 = inline('abs(polyval(a,x) - fx)','a','x','fx');
f73222 = inline('polyval(a,x) - fx','a','x','fx');
N = 2; % the degree of approximating polynomial
a0 = zeros(1,N + 1); %initial guess of polynomial coefficients
xx = -2+[0:200]'/50; %intermediate points
fx = feval(f,xx);    % and their function values f(xx)
ao_m = fminimax(f73221,a0,[],[],[],[],[],[],[],[],xx,fx) %fminimax sol
for n = 1:N+1, C(:,n) = xx.^(N + 1 - n); end
ao_ll = lsqlin(C,fx) %linear LS to minimize (Ca - fx)^2 with no constraint
ao_ln = lsqnonlin(f73222,a0,[],[],[],xx,fx) %nonlinear LS
c2 = cheby(f,N,-2,2) %Chebyshev polynomial over [-2,2]
plot(xx,fx,':', xx,polyval(ao_m,xx),'m', xx,polyval(ao_ll,xx),'r')
hold on, plot(xx,polyval(ao_ln,xx),'b', xx,polyval(c2,xx),'--')
axis([-2 2 -0.4 1.1])
```

7.3.3 Linear Programming (LP)

The linear programming (LP) scheme implemented by the MATLAB built-in routine

`"[xo,fo] = linprog(f,A,b,Aeq,Beq,l,u,x0,options)"`

is designed to solve an LP problem, which is a constrained minimization problem as follows.

$$\text{Min } f(\mathbf{x}) = \mathbf{f}^T\mathbf{x} \tag{7.3.10a}$$

$$\text{subject to} \quad A\mathbf{x} \le \mathbf{b}, \quad A_{eq}\mathbf{x} = \mathbf{b}_{eq}, \quad \text{and} \quad \mathbf{l} \le \mathbf{x} \le \mathbf{u} \tag{7.3.10b}$$

```
%nm733 to solve a Linear Programming problem.
% Min f*x=-3*x(1)-2*x(2) s.t. Ax <= b, Aeq = beq and l <= x <= u
x0 = [0 0]; %initial point
f = [-3 -2]; %the coefficient vector of the objective function
A = [3 4; 2 1]; b = [7; 3]; %the inequality constraint Ax <= b
Aeq = [-3 2]; beq = 2; %the equality constraint Aeq*x = beq
l = [0 0]; u = [10 10]; %lower/upper bound l <= x <= u
[xo_lp,fo_lp] = linprog(f,A,b,Aeq,beq,l,u)
cons_satisfied = [A; Aeq]*xo_lp-[b; beq] %how constraints are satisfied
f733o=inline('-3*x(1)-2*x(2)', 'x');
[xo_con,fo_con] = fmincon(f733o,x0,A,b,Aeq,beq,l,u)
```

It produces the solution (column) vector $\mathbf{x}_o$ and the minimized value of the objective function $f(\mathbf{x}_o)$ as its first and second output arguments `xo` and `fo`, where the objective function and the constraints excluding the constant term are linear in terms of the independent (decision) variables. It works for such linear optimization problems as (7.3.10) more efficiently than the general constrained optimization routine "`fmincon()`".

The usage of the routine "`linprog()`" is exemplified by the MATLAB program "`nm733.m`", which uses the routine for solving an LP problem described as

$$\text{Min } f(\mathbf{x}) = \mathbf{f}^T\mathbf{x} = [-3\ \ -2][x_1\ x_2]^T = -3x_1 - 2x_2 \tag{7.3.11a}$$

s.t.

$$A\mathbf{x} = \begin{bmatrix} -3 & 2 \\ 3 & 4 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{matrix} = \\ \le \\ \le \end{matrix} \begin{bmatrix} 2 \\ 7 \\ 3 \end{bmatrix} = \mathbf{b} \quad \text{and}$$

$$\mathbf{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \le \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 10 \\ 10 \end{bmatrix} = \mathbf{u} \tag{7.3.11b}$$

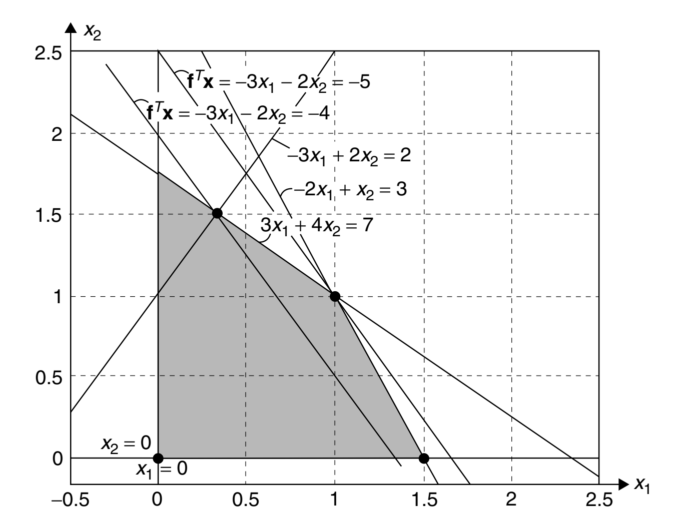


Figure 7.15 The objective function, constraints, and solutions of an LP problem.

Table 7.3 The Names of MATLAB Built-In Minimization Routines in MATLAB 5.x/6.x

Minimization Methods	Unconstrained Minimization			Constrained Minimization				
	Bracketing	Non-Gradient-Based	Gradient-Based	Linear	Nonlinear	Linear LS	Nonlinear LS	Minimax
MATLAB 5.x	fmin	fmins	fminu	lp	constr	leastsq	conls	minimax
MATLAB 6.x	fminbnd	fminsearch	fminunc	linprog	fmincon	lsqnonlin	lsqlin	fminimax

The program also applies the general constrained minimization routine "`fmincon()`" to solve the same problem for cross-check. Readers are welcome to run the program and see the results.

```
>> nm733
    xo_lp = [0.3333  1.5000], fo_lp = -4.0000
    cons_satisfied = -0.0000 % <= 0(inequality)
                     -0.8333 % <= 0(inequality)
                     -0.0000 % = 0(equality)
    xo_con = [0.3333  1.5000], fo_con = -4.0000
```

In this result, the solutions obtained by using the two routines "`linprog()`" and "`fmincon()`" agree with each other, satisfying the inequality/equality constraints and it can be assured by Fig. 7.15.

In Table 7.3, the names of MATLAB built-in minimization routines in MATLAB version 5.x and 6.x are listed.

PROBLEMS

7.1 Modification of Golden Search Method

In fact, the golden search method explained in Section 7.1 requires only one function evaluation per iteration, since one point of a new interval coincides with a point of the previous interval so that only one trial point is updated. In spite of this fact, the MATLAB routine "`opt_gs()`" implementing the method performs the function evaluations twice per iteration. An improvement may be initiated by modifying the declaration type as

```
[xo,fo] = opt_gs1(f,a,e,fe,r1,b,r,TolX,TolFun,k)
```

so that anyone could use the new routine as in the following program, where its input argument list contains another point (`e`) as well as the new end point (`b`) of the next interval, its function value (`fe`), and a parameter (`r1`) specifying if the point is the left one or the right one. Based on this idea, how do you revise the routine "`opt_gs()`" to cut down the number of function evaluations?

```
%nm7p01.m  to perform the revised golden search method
f701 = inline('x.*(x-2)', 'x');
a = 0; b = 3; r = (sqrt(5)-1)/2;
TolX = 1e-4; TolFun = 1e-4; MaxIter=100;
h = b - a; rh = r*h;
c = b - rh; d = a + rh;
fc = f701(c);  fd = f701(d);
if fc < fd, [xo,fo] = opt_gs1(f701,a,c,fc,1 - r,d,r,TolX,TolFun,MaxIter)
 else [xo,fo] = opt_gs1(f701,c,d,fd,r,b,r, TolX,TolFun,MaxIter)
end
```

7.2 Nelder–Mead, Steepest Descent, Newton, SA, GA and `fminunc()`, `fminsearch()`

Consider a two-variable objective function

$$f(\mathbf{x}) = x_1^4 - 12x_1^2 - 4x_1 + x_2^4 - 16x_2^2 - 5x_2 - 20\cos(x_1 - 2.5)\cos(x_2 - 2.9) \quad \text{(P7.2.1)}$$

whose gradient vector function is

$$\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x}) = \begin{bmatrix} 4x_1^3 - 24x_1 - 4 + 20\sin(x_1 - 2.5)\cos(x_2 - 2.9) \\ 4x_2^3 - 32x_2 - 5 + 20\cos(x_1 - 2.5)\sin(x_2 - 2.9) \end{bmatrix} \quad \text{(P7.2.2)}$$

You have the MATLAB functions `f7p02()`, `g7p02()` defining the objective function $f(\mathbf{x})$ and its gradient function $\mathbf{g}(\mathbf{x})$. You also have a part of the MATLAB program which plots a mesh/contour-type graphs for $f(\mathbf{x})$. Note that this gradient function has nine zeros as listed in Table P7.2.1.

Table P7.2.1 Extrema (Maxima/Minima) and Saddle Points of the Function (P7.2.1)

Points	Signs of $\partial^2 f/\partial x_i^2$		Points	Signs of $\partial^2 f/\partial x_i^2$	
(1) [0.6965 −0.1423]	−, −	M	(6) [−1.6926 −0.1183]		
(2) [2.5463 −0.1896]			(7) [−2.6573 −2.8219]	+, +	m
(3) [2.5209 2.9027]	+, +	G	(8) [−0.3227 −2.4257]		
(4) [−0.3865 2.9049]			(9) [2.5216 −2.8946]	+, +	m
(5) [−2.6964 2.9031]					

(a) From the graphs (including Fig. P7.2) which you get by running the (unfinished) program, determine the characteristic of each of the nine points, that is, whether it is a local maximum(M)/minimum(m), the global minimum(G) or a saddle point(S) which is a minimum with respect to one variable and a maximum with respect to another variable. Support your judgment by telling the signs of the second derivatives of $f(\mathbf{x})$ with respect to x_1 and x_2.

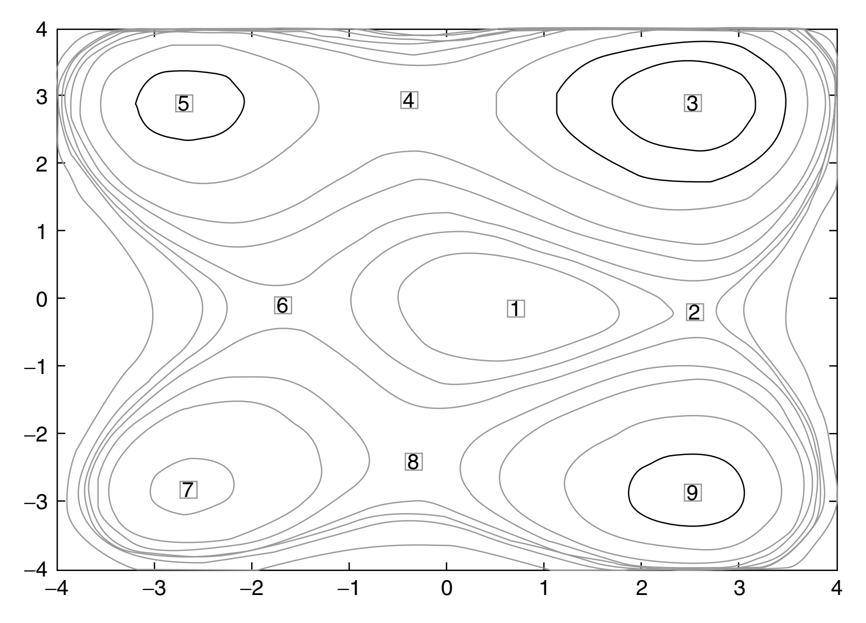


Figure P7.2 The contour, extrema and saddle points of the objective function (P7.2.1).

```
%nm7p02 to minimize an objective ftn f(x) by the Newton method
f = 'f7p02'; g = 'g7p02';
l = [-4 -4]; u = [4 4];
x1 = l(1):.25:u(1); x2 = l(2):.25:u(2); [X1,X2] = meshgrid(x1,x2);
for m = 1:length(x1)
   for n = 1:length(x2), F(n,m) = feval(f,[x1(m) x2(n)]); end
end
figure(1), clf, mesh(X1,X2,F)
figure(2), clf,
contour(x1,x2,F,[-125 -100 -75 -50 -40 -30 -25 -20 0 50])
... ... ... ...
```

```
function y = f7p02(x)
y = x(1)^4 - 12*x(1)^2 - 4*x(1) + x(2)^4 - 16*x(2)^2 - 5*x(2)...
        -20*cos(x(1) - 2.5)*cos(x(2) - 2.9);
```

```
function [df,d2f] = g7p02(x) % the 1st/2nd derivatives
df(1) = 4*x(1)^3 - 24*x(1) - 4 + 20*sin(x(1) - 2.5)*cos(x(2) - 2.9);%(P7.2.2)
df(2) = 4*x(2)^3 - 32*x(2)-5 + 20*cos(x(1) - 2.5)*sin(x(2) - 2.9);%(P7.2.2)
d2f(1) = 12*x(1)^2 - 24 + 20*cos(x(1) - 2.5)*cos(x(2) - 2.9);  %(P7.2.3)
d2f(2) = 12*x(2)^2 - 32 + 20*cos(x(1) - 2.5)*cos(x(2) - 2.9);  %(P7.2.3)
```

$$
\begin{aligned}
\partial^2 f/\partial x_1^2 &= 12x_1^2 - 24 + 20\cos(x_1 - 2.5)\cos(x_2 - 2.9) \\
\partial^2 f/\partial x_2^2 &= 12x_1^2 - 32 + 20\cos(x_1 - 2.5)\cos(x_2 - 2.9)
\end{aligned}
\tag{P7.2.3}
$$

(b) Apply the Nelder–Mead method, the steepest descent method, the Newton method, the simulated annealing (SA), genetic algorithm (GA), and the MATLAB built-in routines `fminunc()`, `fminsearch()` to minimize the objective function (P7.2.1) and fill in Table P7.2.2 with the number and character of the point reached by each method.

Table P7.2.2 Points Reached by Several Optimization Routines

Initial Point	Reached Point						
$\mathbf{x}_0$	Nelder	Steepest	Newton	`fminunc`	`fminsearch`	SA	GA
(0, 0)	(5)/m						
(1, 0)		(3)/G					
(1, 1)			(9)/m				
(0, 1)				(3)/G			
(−1, 1)					(5)/m		
(−1, 0)						≈(3)/G	
(−1, −1)							(3)/G
(0, −1)	(9)/m						
(1, −1)		(9)/m					
(2, 2)			(3)/G				
(−2, −2)				(7)/m			

(c) Overall, the point reached by each minimization algorithm depends on the starting point—that is, the initial value of the independent variable as well as the characteristic of the algorithm. Fill in the blanks in the following sentences. Most algorithms succeed to find the global minimum if only they start from the initial point (,), (,), (,), or (,). An algorithm most possibly goes to the closest local minimum (5) if launched from (,) or (,), and it may go to the closest local minimum (7) if launched from (,) or (,). If launched from (,), it may go to one of the two closest local minima (7) and (9) and if launched from (,), it most possibly goes to the closest local minimum (9). But, the global optimization techniques SA and GA seem to work fine almost regardless of the starting point, although not always.

7.3 Minimization of an Objective Function Having Many Local Minima/Maxima

Consider the problem of minimizing the following objective function

$$\text{Min } f(x) = \sin(1/x)/((x-0.2)^2 + 0.1) \tag{P7.3.1}$$

which is depicted in Fig. P7.3. The graph shows that this function has infinitely many local minima/maxima around $x = 0$ and the global minimum about $x = 0.2$.

(a) Find the solution by using the MATLAB built-in routine "`fminbnd()`". Is it plausible?

(b) With nine different values of the initial guess $x_0 = 0.1, 0.2, \ldots, 0.9$, use the four MATLAB routines "`opt_Nelder()`", "`opt_steep()`", "`fminunc()`", and "`fminsearch()`" to solve the problem. Among those 36 tryouts, how many times have you got the right solution?

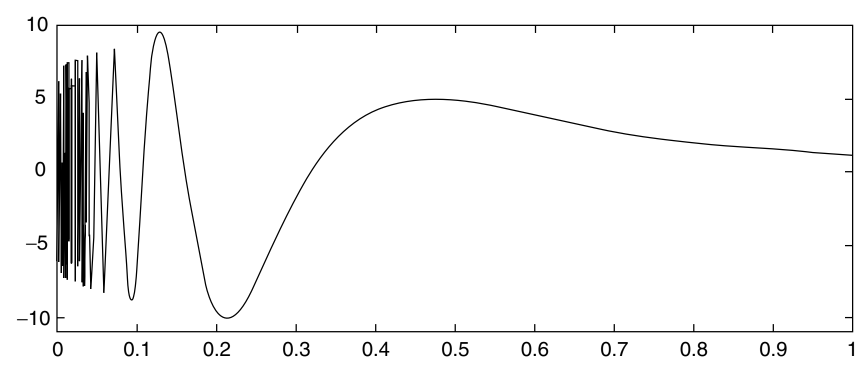


Figure P7.3 The graph of $f(x) = \sin(1/x)/((x - 0.2)^2 + 0.1)$ having many local minima/maxima.

(c) With the values of the parameters set to $l = 0, u = 1, q = 1, \varepsilon_f = 10^{-9}$, $k_{\max} = 1000$ and the initial guess $x_0 = 0.1, 0.2, \ldots, 0.9$, use the SA (simulated annealing) routine "`sim_anl()`" to solve the problem. You can test the performance of the routine and your luck by running the routine four times for the same problem and finding the probability of getting the right solution.

(d) With the values of the parameters set to $l = 0$, $u = 1$, $N_p = 30$, $N_b = 12$, $P_c = 0.5$, $P_m = 0.01$, $\eta = 1$, $k_{\max} = 1000$ and the initial guess $x_0 = 0.1, 0.2, \ldots, 0.9$, use the GA (genetic algorithm) routine "`genetic()`" to solve the problem. As in (c), you can run the routine four times for the same problem and find the probability of getting the right solution in order to test the performance of the routine and your luck.

7.4 Linear Programming Method

Consider the problem of maximizing a linear objective function

$$\text{Max } f(\mathbf{x}) = \mathbf{f}^T\mathbf{x} = [\,3 \quad 2 \quad -1\,][\,x_1 \quad x_2 \quad x_3\,]^T \tag{P7.4.1a}$$

subject to the constraints

$$A\mathbf{x} = \begin{bmatrix} 3 & -2 & 0 \\ -3 & -4 & 0 \\ -2 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \begin{matrix} = \\ \geq \\ \geq \end{matrix} \begin{bmatrix} -2 \\ -7 \\ -3 \end{bmatrix} = \mathbf{b} \quad \text{and}$$

$$\mathbf{l} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \leq \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix} = \mathbf{u} \tag{P7.4.1b}$$

Jessica is puzzled with this problem, which is not a minimization but a maximization. How do you suggest her to solve it? Make the program that uses the MATLAB built-in routines "`linprog()`" and "`fmincon()`" to solve this problem and run it to get the solutions.

7.5 Constrained Optimization and Penalty Method

Consider the problem of minimizing a nonlinear objective function

$$\text{Min}_{\mathbf{x}} f(\mathbf{x}) = -3x_1 - 2x_2 + M(3x_1 - 2x_2 + 2)^2 \qquad \text{(P7.5.1a)}$$

$$(M : \text{a large positive number})$$

subject to the constraints

$$\begin{bmatrix} 3 & 4 \\ -2 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{matrix} \le \\ \ge \end{matrix} \begin{matrix} 7 \\ -3 \end{matrix} \quad \text{and} \quad \mathbf{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \le \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 10 \\ 10 \end{bmatrix} = \mathbf{u} \qquad \text{(P7.5.1b)}$$

(a) With the two values of the weighting factor $M = 20$ and 10,000 in the objective function (P7.5.1a), apply the MATLAB built-in routine "`fmincon()`" to find the solutions to the above constrained minimization problem. In order to do this job, you might have to make the variable parameter M passed to the objective function (defined in an M-file) either through "`fmincon()`" or directly by declaring the parameter as global both in the main program and in the M-file defining (P7.5.1a). In case you are going to have the parameter passed through "`fmincon()`" to the objective function, you should have the parameter included in the input argument list of the objective function as

```
function f=f7p05M(x,M)
f = -3*x(1)-2*x(2)+M*(3*x(1)-2*x(2)+2).^2;
```

Additionally, you should give empty matrices (`[]`) as the ninth input argument (for a nonlinear inequality/equality constraint function '`nonlcon`') as well as the 10th one (for '`options`') and the value of M as the 11th one of the routine "`fmincon()`".

```
xo = fmincon('f7p05M',x0,A,b,[],[],l,u,[],[],M)
```

For reference, type '`help fmincon`' into the MATLAB command window.

(b) Noting that the third (squared) term of the objective function (P7.5.1a) has its minimum value of zero for $3x_1 - 2x_2 + 2 = 0$ and, thus, it actually represents the penalty (Section 7.2.2) imposed for not satisfying the equality constraint

$$3x_1 - 2x_2 + 2 = 0 \qquad \text{(P7.5.2)}$$

tell which of the solutions obtained in (a) is more likely to satisfy this constraint and support your answer by comparing the values of the left-hand side of this equality for the two solutions.

(c) Removing the third term from the objective function and splitting the equality constraint into two reversed inequality constraints, we can modify the problem as follows:

$$\text{Min}_{\mathbf{x}} f(\mathbf{x}) = -3x_1 - 2x_2 \qquad \text{(P7.5.3a)}$$

subject to the constraints

$$\begin{bmatrix} 3 & 4 \\ -2 & -1 \\ 3 & -2 \\ 3 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{matrix} \le 7 \\ \ge -3 \\ \le -2 \\ \ge -2 \end{matrix} \quad \text{and} \qquad \text{(P7.5.3b)}$$

$$\mathbf{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \le \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 10 \\ 10 \end{bmatrix} = \mathbf{u}$$

Noting that this fits the linear programming, apply the routine "`linprog()`" to solve this problem.

(d) Treating the equality constraint separately from the inequality constraints, we can modify the problem as follows:

$$\text{Min}_{\mathbf{x}} f(\mathbf{x}) = -3x_1 - 2x_2 \qquad \text{(P7.5.4a)}$$

subject to the constraints

$$\begin{bmatrix} 3 & -2 \\ 3 & 4 \\ -2 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{matrix} = -2 \\ \le \ \ 7 \\ \ge -3 \end{matrix} \quad \text{and} \quad \mathbf{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \le \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 10 \\ 10 \end{bmatrix} = \mathbf{u} \qquad \text{(P7.5.4b)}$$

Apply the two routines "`linprog()`" and "`fmincon()`" to solve this problem and see if the solutions agree with the solution obtained in (c).

(cf) Note that, in comparison with the routine "`fmincon()`", which can solve a general nonlinear optimization problem, the routine "`linprog()`" is made solely for dealing with a class of optimization problems having a linear objective function with linear constraints.

7.6 Nonnegative Constrained LS and Constrained Optimization

Consider the problem of minimizing a nonlinear objective function

$$\text{Min}_{\mathbf{x}} \ \|C\mathbf{x} - \mathbf{d}\|^2 = [C\mathbf{x} - \mathbf{d}]^T [C\mathbf{x} - \mathbf{d}] \qquad \text{(P7.6.1a)}$$

subject to the constraints

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \ge \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \mathbf{l} \qquad \text{(P7.6.1b)}$$

where

$$C = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 1 \end{bmatrix}, \qquad \mathbf{d} = \begin{bmatrix} 5.1 \\ 10.8 \\ 6.8 \end{bmatrix} \qquad \text{(P7.6.1c)}$$

(a) Noting that this problem has no other constraints than the lower bound, apply the constrained linear least-squares routine "`lsqlin()`" to find the solution.

(b) Noting that the lower bounds for all the variables are zeros, apply the MATLAB built-in routine "`lsqnonneg()`" to find the solution.

(c) Apply the general-purpose constrained optimization routine "`fmincon()`" to find the solution.

7.7 Constrained Optimization Problems

Solve the following constrained optimization problems by using the MATLAB built-in routine "`fmincon()`".

(a) $$\text{Min}_{\mathbf{x}}\, x_1^3 - 5x_1^2 + 6x_1 + x_2^2 - 2x_2 + x_3 \quad \text{(P7.7.1a)}$$

subject to the constraints

$$\begin{matrix} x_1^2 + x_2^2 - x_3 \le 0 \\ x_1^2 + x_2^2 + x_3^2 \ge 6 \\ x_3 \le 5 \end{matrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \ge \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{l} \quad \text{(P7.7.1b)}$$

Try the routine "`fmincon()`" with the initial guesses listed in Table P7.7.

Table P7.7 The Results of Applying "`fmincon()`" with Different Initial Guess

	Initial Guess $\mathbf{x}_0$	Lower Bound	$\mathbf{x}^o$	$f(\mathbf{x}^o)$	Remark (warning ?)
(a)	[0 0 0]	0			No feasible solution (w)
	[1 1 5]	0			Not a minimum
	[0 0 5]	0			Minimum
	[1 0 2]	0	[1.29 0.57 2]	2.74	
(b1)	[0 0 0]	0	[0 0 0]	0	
	[10 10 10]	0			Maximum (good)
(b2)	[0 0 0]	0			No feasible solution (w)
	[10 10 10]	0			Not a minimum, but the max
	[0.1 0.1 3]	0			One of many minima (w)
(c1)	[0 0 0]	0			
	[0.1 0.1 0.1]	0	[1 1 1]	3	Maximum (good)
	[0 1 2]	0			
(c2)	[0 0 0]	0	[1 1 1]	3	Not a minimum, but the max
	[0.1 0.1 0.1]	0			
	[0 1 2]	0			One of many minima
(d)	[1.0 0.5]	0			Weird (warning)
	[0.2 0.3]	0	[10.25 0]	∞	
	[2 5]	0	[5.77 8.17]	25.98	
	[100 10]	0			Minimum

(b1) $\text{Max}_{\mathbf{x}}\ x_1x_2x_3$ (P7.7.2a)

subject to the constraints

$$x_1x_2 + x_2x_3 + x_3x_1 = 3 \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{(P7.7.2b)}$$

Try the routine "`fmincon()`" with the initial guesses listed in Table P7.7.

(b2) $\text{Min}_{\mathbf{x}}\ x_1x_2x_3$ (P7.7.3a)

subject to the constraints (P7.7.2b).

Try the routine "`fmincon()`" with the initial guesses listed in Table P7.7.

(c1) $\text{Max}_{\mathbf{x}}\ x_1x_2 + x_2x_3 + x_3x_1$ (P7.7.4a)

subject to the constraints

$$x_1 + x_2 + x_3 = 3 \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{(P7.7.4b)}$$

Try the routine "`fmincon()`" with the initial guesses listed in Table P7.7.

(c2) $\text{Min}_{\mathbf{x}}\ x_1x_2 + x_2x_3 + x_3x_1$ (P7.7.5a)

subject to the constraints (P7.7.4b).

Try the routine "`fmincon()`" with the initial guesses listed in Table P7.7.

(d) $\text{Min}_{\mathbf{x}}\ \dfrac{10000}{x_1x_2^2}$ (P7.7.6a)

subject to the constraints

$$x_1^2 + x_2^2 = 100 \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} > \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \text{(P7.7.6b)}$$

Try the routine "`fmincon()`" with the initial guesses listed in Table P7.7.

(e) Does the routine work well with all the initial guesses? If not, does it matter whether the starting point is inside the admissible region?

(cf) Note that, in order to solve the maximization problem by "`fmincon()`", we have to reverse the sign of the objective function. Note also that the objective functions (P7.7.3a) and (P7.7.5a) have infinitely many minima having the value $f(\mathbf{x}) = 0$ in the admissible region satisfying the constraints.

(cf) One might be disappointed with the reliability of the MATLAB optimization routines to see that they may fail to find the optimal solution depending on the initial guess. But, how can a human work be perfect in this world? It implies the difficulty of nonlinear constrained optimization problems and can never impair the celebrity and reliability of MATLAB. Actually, it demonstrates the importance of studying some numerical stuff in addition to just getting used to the various MATLAB commands and routines.

Here is a tip for the usage of "`fmincon()`": it might be better to use with an initial guess that is not at the origin, but in the admissible region satisfying the constraints, even though it does not guarantee the success of the routine. It might also be helpful to apply the routine with several values of the initial guess and then choose the best result.

7.8 Constrained Optimization and Penalty Method

Consider again the constrained minimization problem having the objective function (E7.3.1a) and the constraints (E7.3.1b).

$$\text{Min } f(\mathbf{x}) = \{(x_1 + 1.5)^2 + 5(x_2 - 1.7)^2\}\{(x_1 - 1.4)^2 + 0.6(x_2 - 0.5)^2\} \tag{P7.8.1a}$$

$$\text{s.t.} \quad \mathbf{g}(\mathbf{x}) = \begin{bmatrix} -x_1 \\ -x_2 \\ 3x_1 - x_1x_2 + 4x_2 - 7 \\ 2x_1 + x_2 - 3 \\ 3x_1 - 4x_2^2 - 4x_2 \end{bmatrix} \le \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{P7.8.1b}$$

In Example 7.3, we made the MATLAB program "`nm722.m`" to solve the problem and defined the objective function (E7.3.2a) having the penalized constraint terms in the file named "f722p.m".

$$\text{Min } l(\mathbf{x}) = \{(x_1 + 1.5)^2 + 5(x_2 - 1.7)^2\}\{(x_1 - 1.4)^2 + 0.6(x_2 - 0.5)^2\} + \sum_{m=1}^{5} v_m \psi_m(g_m(\mathbf{x})) \tag{P7.8.2a}$$

where

$$\psi_m(g_m(\mathbf{x})) = \begin{cases} 0 & \text{if } g_m(\mathbf{x}) \le 0 \text{ (constraint satisfied)} \\ \exp(e_m g_m(\mathbf{x})) & \text{if } g_m(\mathbf{x}) > 0 \text{ (constraint viloated)} \end{cases}$$
$$\text{with } e_m = 1 \ \forall m = 1, \ldots, 5 \tag{P7.8.2b}$$

(a) What is the weighting coefficient vector `v` in the file named "`f722p.m`"? Do the points reached by the routines "`fminsearch()`"/"`opt_steep()`"/"`fminunc()`" satisfy all the constraints so that they are in the admissible region? If not, specify the constraint(s) violated by the points.

(b) Suppose the fourth constraint was violated by the point in (a). Then, how would you modify the weighting coefficient vector `v` so that the violated constraint can be paid more respect? Choose one of the following two weighting coefficient vectors:

(i) `v = [1 1 1 1/3 1]`

(ii) `v = [1 1 1 3 1]`

and modify the file "`f722p.m`" with this coefficient vector. Then, run the program "`nm722.m`", fill in the 22 blanks of Table P7.8 with the results and see if the fourth constraint is still violated by the points reached by the optimization routines?

(c) Instead of the penalty method, apply the intrinsically constrained optimization routine "`fmincon()`" with the initial guesses $\mathbf{x}_0 = [0.4\ 0.5]$ and [0.2 4] to solve the problem described by Eq. (E7.3.1) or (P7.8.1) and fill in Table P7.8 with the results concerning the reached point and the corresponding values of the objective/constraint functions.

(d) Based on the results listed in Table P7.8, circle the right word in each of the parentheses in the following sentences:

- For penalty methods, the non-gradient-based minimization routines like "`Nelder()`"/"`fminsearch()`" may work (better, worse) than the gradient-based minimization routines like "`opt_steep()`"/"`fminunc()`".
- If some constraint is violated, you had better (increase, decrease) the corresponding weight coefficient.

(cf) Besides, unconstrained optimization with the penalized constraints in the objective function sometimes works better than the constrained optimization routine "`fmincon()`".

Table P7.8 The Results of Penalty Methods Depending on the Initial Guess and Weighting Factor

v		The Starting Point $\mathbf{x}_0 = [0.4\quad 0.5]$					$\mathbf{x}_0 = [0.2\quad 4]$				
		`Nelder`	`fminsearch`	`steep`	`fminunc`	`fmincon`	`Nelder`	`fminsearch`	`steep`	`fminunc`	`fmincon`
1	$\mathbf{x}^o$	1.21 0.58		1.34 0.62			1.34 0.62		1.34 0.62		
1	$\mathbf{f}^o$	0.53		0.17			0.17		0.17		
1 1/3 1	$\mathbf{c}^o$	−1.21 −0.58 −1.76 −0.00 −0.00	−1.34 −0.62 −1.34 −0.00	−1.34 −0.62 −1.34 −0.00	−1.38 −0.63 −1.19 −0.00	−1.21 −0.58 −1.76 0.00 0.00	−1.34 −0.62 −1.34 0.29 −0.00	−1.34 −0.62 −1.34 −0.00	−1.34 −0.62 −1.33 −0.00	1.26 −1.70 −1.84 −3.82 −22.1	0.00 −1.59 −0.65 −1.41 −16.4
1	$\mathbf{x}^o$	1.21 0.58	1.21 0.58	1.12 0.76	1.18 0.64	— —	1.21 0.58	1.21 0.58	1.15 0.71	−1.26 1.70	— —
1	f^o	0.53	0.53	1.36	0.79	—	0.53	0.53	1.08	0.46	—
1 3 1	$\mathbf{c}^o$	−1.21 −0.58 −1.76 −0.00 −0.00	−1.21 −0.58 −1.76 −0.00	−1.12 −0.76 −1.44 −2.04	−1.18 −0.64 −1.65 −0.70	—	−1.21 −0.58 −1.76 −0.00 −0.00	−1.21 −0.58 −1.76 −0.00	−1.15 −0.71 −1.54 −1.39	1.26 −1.70 −1.84 −3.82 −22.1	—

7.9 A Constrained Optimization on Location

A company has three factories that are located at the points (−16,4), (6,5), and (3,−9), respectively, in the x_1x_2-plane, and the numbers of deliveries to those factories are 5, 6, and 10 per month, respectively (Fig. P7.9). The company has a plan to build a new warehouse in its site bounded by

$$|x_1 - 1| + |x_2 - 1| \leq 2 \tag{P7.9.1}$$

and is trying to minimize the monthly mileage of delivery trucks in determining the location of a new warehouse on the assumption that the distance between two points represents the driving distance.

(a) What is the objective function that must be defined in the program "`nm7p09.m`"?

(b) What is the statement defining the inequality constraint (P7.9.1)?

(c) Complete and run the program "`nm7p09.m`" to get the optimum location of the new warehouse.

```
function [C,Ceq] = fp_warehouse_c(x)
C = sum(abs(x - [1 1])) - 2;
Ceq = []; % No equality constraint
```

```
%nm7p09.m to solve the warehouse location problem
f = 'sqrt([sum((x - [-16 4]).^2) sum((x - [6 5]).^2) sum((????????).^2)])';
fp_warehouse = inline([f '*[?;?;?]'],'x');
x0 = [1 1]; A = []; b = []; Aeq = []; beq = []; l = []; u = [];
xo = fmincon(fp_warehouse,x0,A,b,Aeq,beq,l,u,'fp_warehouse_c')
```

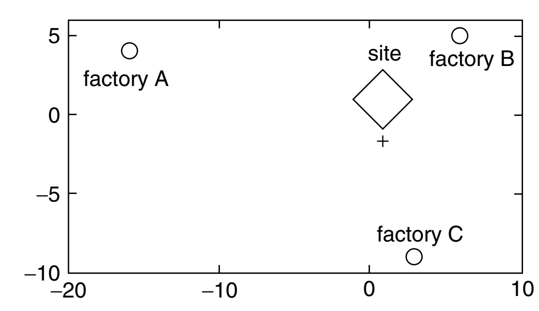


Figure P7.9 The site of a new warehouse and the locations of the factories.

7.10 A Constrained Optimization on Ray Refraction

A light ray follows the path that takes the shortest time when it travels in the space. We want to find the three angles θ_1,θ_2, and θ_3 (measured between the array and the normal to the material surface) of a ray traveling from $P = (0, 0)$ to $Q = (L, -(d_1 + d_2 + d_3))$ through a transparent material of thickness d_2 and index of refraction n as depicted in Fig. P7.10. Note the following things.

- Since the speed of light in the transparent material is $v = c/n$ (c is the speed of light in the free space), the traveling time to be minimized can be expressed as

$$\text{Min } f(\boldsymbol{\theta}, \mathbf{d}, n, L) = \frac{d_1}{c\cos\theta_1} + \frac{nd_2}{c\cos\theta_2} + \frac{d_3}{c\cos\theta_3} \tag{P7.10.1}$$

- The sum of the three horizontal distances traveled by the light ray must be L:

$$g(\boldsymbol{\theta}, \mathbf{d}, n, L) = \sum_{i=1}^{3} d_i \tan\theta_i - L = 0 \tag{P7.10.2}$$

- The horizontal distance L and the index of refraction n are additionally included in the input argument lists of both the objective function $f(\boldsymbol{\theta}, \mathbf{d}, n, L)$ and the constraint function $g(\boldsymbol{\theta}, \mathbf{d}, n, L)$ regardless of whether or not they are used in each function. It is because the objective function and the constraint function of the MATLAB routine "`fmincon()`" must have the same input arguments.

(a) Compose a program "`nm7p10a.m`" that solves the above constrained minimization problem to find the three angles θ_1, θ_2, and θ_3 for $n = 1.52$, $d_1 = d_2 = d_3 = 1$[cm], and different values of $L = 0.6$:0.3:6 and plots sin(θ_1)/sin(θ_2) and sin(θ_3)/sin(θ_2) versus L.

(b) Compose a program "`nm7p10b.m`" that finds the three angles θ_1,θ_2, and θ_3 for $L = 3$ cm, $d_1 = d_2 = d_3 = 1$ cm, and different values of $n = 1$:0.01:1.6 and plots sin(θ_1)/sin(θ_2) and sin(θ_3)/sin(θ_2) versus n.

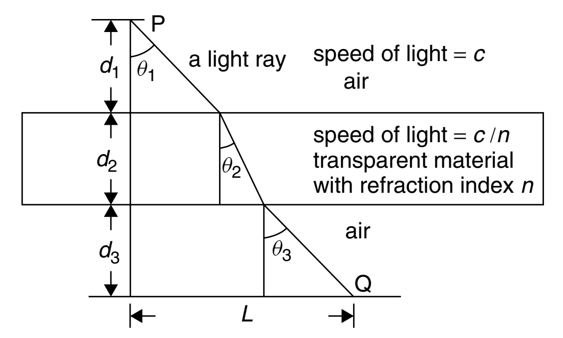


Figure P7.10 Refraction of a light ray at an air–glass interface.

7.11 A Constrained Optimization on OFDM System

In order to find the average modulation order x_i for each user of an OFDM (orthogonal frequency division multiplex) system that has N(128) subchannels to assign to each of the four users in the environment of noise power

N_0 and the bit error rate (probability of bit error) P_e, Seung-hee, a communication system expert, formulated the following constrained minimization problem:

$$\text{Min } f(x) = \sum_{i=1}^{4} (2^{x_i} - 1)\frac{N_0}{3}2(\text{erfc}^{-1}(P_e/2))^2 \frac{a_i}{x_i} \qquad \text{(P7.11.1)}$$

subject to

$$g(x) = \sum_{i=1}^{4} \frac{a_i}{x_i} - N = 0 \qquad \text{(P7.11.2)}$$

$$\text{with } N = 128, \text{ and } a_i\text{: the data rate of each user}$$

where $\text{erfc}^{-1}(x)$ is the inverse function of the complementary error function defined by Eq. (P4.9.3) and is installed as the MATLAB built-in function '`erfcinv()`'. He defined the objective function and the constraint function as below and save them in the M-files named "`fp_bits1.m`" and "`fp_bits_c.m`".

```
function y = fp_bits1(x,a,N,Pe)
N0 = 1; y = sum((2.^x-1)*N0/3*2*erfcinv(Pe/2).^2.*a./x);
```

```
function [C,Ceq] = fp_bits_c(x,a,N,Pe)
C = []; Ceq = sum(a./x) - N;
```

Compose a program that solves the above constrained minimization problem (with $N_0 = 1$ and $P_e = 10^{-4}$) to get the modulation order x_i of each user for five different sets of data rates

$$a = [32\ 32\ 32\ 32], [64\ 32\ 32\ 32], [128\ 32\ 32\ 32], [256\ 32\ 32\ 32], \text{ and } [512\ 32\ 32\ 32]$$

and plots a_1/x_1(the number of subchannels assigned to user 1) versus a_1 (the data rate of user 1). If you feel uneasy about the results obtained with your initial guesses, try with the initial guesses as follows for each set of data rates, respectively:

$$x_0 = [0.5\ 0.5\ 0.5\ 0.5], [1\ 1\ 1\ 1], [1\ 1\ 1\ 1], [2\ 2\ 2\ 2], \text{ and } [4\ 4\ 4\ 4]$$

8

MATRICES AND EIGENVALUES

In this chapter, we will look at the eigenvalue or characteristic value λ and its corresponding eigenvector or characteristic vector $\mathbf{v}$ of a matrix.

8.1 EIGENVALUES AND EIGENVECTORS

The eigenvalue or characteristic value and its corresponding eigenvector or characteristic vector of an $N \times N$ matrix A are defined as a scalar λ and a nonzero vector $\mathbf{v}$ satisfying

$$A\mathbf{v} = \lambda\mathbf{v} \Leftrightarrow (A - \lambda I)\,\mathbf{v} = \mathbf{0}\,(\mathbf{v} \neq \mathbf{0}) \tag{8.1.1}$$

where $(\lambda, \mathbf{v})$ is called an eigenpair and there are N eigenpairs for the $N \times N$ matrix A.

How do we get them? Noting that

- in order for the above equation to hold for any nonzero vector $\mathbf{v}$, the matrix $[A - \lambda I]$ should be singular—that is, its determinant should be zero ($|A - \lambda I| = 0$)— and
- the determinant of the matrix $[A - \lambda I]$ is a polynomial of degree N in terms of λ,

we first must find the eigenvalue λ_i's by solving the so-called characteristic equation

$$|A - \lambda I| = \lambda^N + a_{N-1}\lambda^{N-1} + \cdots + a_1\lambda + a_0 = 0 \tag{8.1.2}$$

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

and then substitute the λ_i's, one by one, into Eq. (8.1.1) to solve it for the eigenvector $\mathbf{v}_i$'s. This is, however, not always so simple, especially if some root (eigenvalue) of Eq. (8.1.2) has multiplicity $k > 1$, since we have to generate k independent eigenvectors satisfying Eq. (8.1.1) for such an eigenvalue. Still, we do not have to worry about this, thanks to the MATLAB built-in routine "`eig()`", which finds us all the eigenvalues and their corresponding eigenvectors for a given matrix. How do we use it? All we need to do is to define the matrix, say A, and type a single statement into the MATLAB command window as follows.

```
>>[V,Lambda] = eig(A) %e = eig(A) just for eigenvalues
```

Let us take a look at the following example.

Example 8.1. Eigenvalues/Eigenvectors of a Matrix.
Let us find the eigenvalues/eigenvectors of the matrix

$$A = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \tag{E8.1.1}$$

First, we find its eigenvalues as

$$|A - \lambda I| = \left| \begin{bmatrix} -\lambda & 1 \\ 0 & -1-\lambda \end{bmatrix} \right| = \lambda^2 + \lambda = 0$$
$$\lambda(\lambda+1) = 0, \qquad \lambda_1 = 0, \lambda_2 = -1 \tag{E8.1.2}$$

and then, get the corresponding eigenvectors as

$$[A - \lambda_1 I]\mathbf{v}_1 = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix} = \begin{bmatrix} v_{21} \\ -v_{21} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$
$$v_{21} = 0, \qquad \mathbf{v}_1 = \begin{bmatrix} v_{11} \\ v_{21} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{E8.1.3a}$$
$$[A - \lambda_2 I]\mathbf{v}_2 = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_{12} \\ v_{22} \end{bmatrix} = \begin{bmatrix} v_{12} + v_{22} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$
$$v_{12} = -v_{22}, \qquad \mathbf{v}_2 = \begin{bmatrix} v_{12} \\ v_{22} \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix} \tag{E8.1.3b}$$

where we have chosen v_{11}, v_{12}, and v_{22} so that the norms of the eigenvectors become one.

Alternatively, we can use the MATLAB command "`eig(A)`" for finding eigenvalues/eigenvectors or "`roots(poly(A))`" just for finding eigenvalues as the roots of the characteristic equation as illustrated by the program "`nm811.m`".

```
%nm811  to get the eigenvalues & eigenvectors of a matrix A.
clear
A = [0 1;0 -1];
[V,L] = eig(A) %V = modal matrix composed of eigenvectors
% L = diagonal matrix with eigenvalues on its diagonal
e = eig(A), roots(poly(A)) %just for eigenvalues
L = V^ - 1*A*V %diagonalize through similarity transformation
     % into a diagonal matrix having the eigenvalues on diagonal.
```

8.2 SIMILARITY TRANSFORMATION AND DIAGONALIZATION

Premultiplying a matrix A by P^{-1} and post-multiplying it by P makes a similarity transformation

$$A \quad \rightarrow \quad P^{-1}AP \tag{8.2.1}$$

Remark 8.1 tells us how a similarity transformation affects the eigenvalues/eigenvectors.

Remark 8.1. Effect of Similarity Transformation on Eigenvalues/Eigenvectors

1. The eigenvalues are not changed by a similarity transformation.

$$|P^{-1}AP - \lambda I| = |P^{-1}AP - P^{-1}\lambda IP| = |P^{-1}||A - \lambda I||P| = |A - \lambda I| \tag{8.2.2}$$

2. Substituting $\mathbf{v} = P\mathbf{w}$ into Eq. (8.1.1) yields

$$A\mathbf{v} = \lambda\mathbf{v}, \qquad AP\mathbf{w} = \lambda P\mathbf{w} = P\lambda\mathbf{w}, \qquad [P^{-1}AP]\mathbf{w} = \lambda\mathbf{w}$$

   This implies that the matrix $P^{-1}AP$ obtained by a similarity transformation has $\mathbf{w} = P^{-1}\mathbf{v}$ as its eigenvector if $\mathbf{v}$ is an eigenvector of the matrix A.

In order to understand the diagonalization of a matrix into a diagonal matrix (having its eigenvalues on the main diagonal) through a similarity transformation, we have to know the following theorem:

Theorem 8.1. Distinct Eigenvalues and Independent Eigenvectors.
If the eigenvalues of a matrix A are all distinct—that is, different from each other—then the corresponding eigenvectors are independent of each other and, consequently, the modal matrix composed of the eigenvectors as columns is nonsingular.

Now, for an $N \times N$ matrix A whose eigenvalues are all distinct, let us put all of the equations (8.1.1) for each eigenvalue-eigenvector pair together to write

$$A[\mathbf{v}_1\ \mathbf{v}_2\ \cdots\ \mathbf{v}_N] = [\mathbf{v}_1\ \mathbf{v}_2\ \cdots\ \mathbf{v}_N]\begin{bmatrix} \lambda_1 & 0 & \cdot & 0 \\ 0 & \lambda_2 & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \lambda_N \end{bmatrix}, \qquad AV = V\Lambda \tag{8.2.3}$$

Then, noting that the modal matrix V is nonsingular and invertible by Theorem 8.1, we can premultiply the above equation by the inverse modal matrix V^{-1} to get

$$V^{-1}AV = V^{-1}V\Lambda \equiv \Lambda \tag{8.2.4}$$

This implies that the modal matrix composed of the eigenvectors of a matrix A is the similarity transformation matrix that can be used for converting the matrix A into a diagonal matrix having its eigenvalues on the main diagonal. Here is an example to illustrate the diagonalization.

Example 8.2. Diagonalization Using the Modal Matrix.

Consider the matrix given in the previous example.

$$A = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \tag{E8.2.1}$$

We can use the eigenvectors (E8.1.3) (obtained in Example 8.1) to construct the modal matrix as

$$V = [\mathbf{v}_1 \ \mathbf{v}_2] = \begin{bmatrix} 1 & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} \end{bmatrix} \tag{E8.2.2}$$

and use this matrix to make a similarity transformation of the matrix A as

$$\begin{aligned} V^{-1}AV &= \begin{bmatrix} 1 & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} \end{bmatrix}^{-1} \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ 0 & -\sqrt{2} \end{bmatrix} \begin{bmatrix} 0 & -1/\sqrt{2} \\ 0 & 1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \end{aligned} \tag{E8.2.3}$$

which is a diagonal matrix having the eigenvalues on its main diagonal.

This job can be performed by the last statement of the MATLAB program "`nm811.m`".

This diagonalization technique can be used to decouple an N-dimensional vector differential equation so that it can be as easy to solve as N independent scalar differential equations. Here is an illustration.

Example 8.3. Decoupling of a Vector Equation Through Diagonalization

(a) For the linear time-invariant (LTI) state equation (6.5.3)

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_s(t) \tag{E8.3.1}$$

$$\text{with } \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ and } u_s(t) = 1 \ \forall \ t \geq 0$$

$\mathbf{x}'(t) = A\mathbf{x}(t) + Bu(t)$ with the initial state $\mathbf{x}(0)$ and the input $u(t)$

we use the modal matrix obtained as (E8.2.2) in Example 8.2 to make a substitution of variable

$$\mathbf{x}(t) = V\mathbf{w}(t), \qquad \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} 1 & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} w_1(t) \\ w_2(t) \end{bmatrix} \tag{E8.3.2}$$

which converts Eq. (E8.3.1) into

$$V\mathbf{w}'(t) = AV\mathbf{w}(t) + Bu_s(t) \tag{E8.3.3}$$

We premultiply (E8.3.3) by V^{-1} to write it in a decoupled form as

$$\mathbf{w}'(t) = V^{-1}AV\mathbf{w}(t) + V^{-1}Bu_s(t) = \Lambda\mathbf{w}(t) + V^{-1}Bu_s(t) \text{ with } \mathbf{w}(0) = V^{-1}\mathbf{x}(0);$$

$$\begin{bmatrix} w_1{}'(t) \\ w_2{}'(t) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} w_1(t) \\ w_2(t) \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 0 & -\sqrt{2} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_s(t) = \begin{bmatrix} u_s(t) \\ -w_2(t) - \sqrt{2}u_s(t) \end{bmatrix} \tag{E8.3.4}$$

$$\text{with } \begin{bmatrix} w_1(0) \\ w_2(0) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & -\sqrt{2} \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ \sqrt{2} \end{bmatrix}$$

where there is no correlation between the variables $w_1(t)$ and $w_2(t)$. Then we can solve these two equations separately to have

$$w_1{}'(t) = u_s(t) \text{ with } w_1(0) = 0;$$

$$sW_1(s) - w_1(0) = \frac{1}{s}; \; W_1(s) = \frac{1}{s^2}; \; w_1(t) = t\ u_s(t) \tag{E8.3.5a}$$

$$w_2{}'(t) = -w_2(t) - \sqrt{2}u_s(t) \text{ with } w_2(0) = \sqrt{2};$$

$$sW_2(s) - w_2(0) = -W_2(s) - \frac{\sqrt{2}}{s};$$

$$W_2(s) = \frac{w_2(0)}{s+1} - \frac{\sqrt{2}}{s(s+1)} = -\frac{\sqrt{2}}{s} + \frac{2\sqrt{2}}{s+1},$$

$$w_2(t) = \sqrt{2}(-1 + 2e^{-t})u_s(t) \tag{E8.3.5b}$$

and substitute this into Eq. (E8.3.2) to get

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} 1 & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} w_1(t) \\ w_2(t) \end{bmatrix} = \begin{bmatrix} 1 & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} t \\ \sqrt{2}(-1 + 2e^{-t}) \end{bmatrix} u_s(t)$$

$$= \begin{bmatrix} t - 1 + 2e^{-t} \\ 1 - 2e^{-t} \end{bmatrix} u_s(t) \tag{E8.3.6}$$

This is the same result as Eq. (6.5.10) obtained in Section 6.5.1.

(b) Suppose Eq. (E8.3.1) has no input term and so we can expect only the natural response resulting from the initial state, but no forced response caused by the input.

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \quad \text{with} \quad \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{E8.3.7}$$

We apply the diagonalization/decoupling method for this equation to get

$$\begin{bmatrix} w_1'(t) \\ w_2'(t) \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} w_1(t) \\ w_2(t) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} w_1(t) \\ w_2(t) \end{bmatrix}$$

$$\text{with } \mathbf{w}(0) = V^{-1}\mathbf{x}(0), \quad \begin{bmatrix} w_1(0) \\ w_2(0) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & -\sqrt{2} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ -\sqrt{2} \end{bmatrix}$$

$$\begin{bmatrix} w_1(t) \\ w_2(t) \end{bmatrix} = \begin{bmatrix} w_1(0)e^{\lambda_1 t} \\ w_2(0)e^{\lambda_2 t} \end{bmatrix} = \begin{bmatrix} 2 \\ -\sqrt{2}e^{-t} \end{bmatrix} \tag{E8.3.8}$$

$$\mathbf{x}(t) \overset{(E8.3.2)}{=} V\mathbf{w}(t) = [\mathbf{v}_1 \ \mathbf{v}_2] \begin{bmatrix} w_1(0)e^{\lambda_1 t} \\ w_2(0)e^{\lambda_2 t} \end{bmatrix} = w_1(0)e^{\lambda_1 t}\mathbf{v}_1 + w_2(0)e^{\lambda_2 t}\mathbf{v}_2$$

$$= \begin{bmatrix} 1 & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 2 \\ -\sqrt{2}e^{-t} \end{bmatrix} = \begin{bmatrix} 2 - e^{-t} \\ e^{-t} \end{bmatrix} \tag{E8.3.9}$$

As time goes by, this solution converges and so the continuous-time system turns out to be stable, thanks to the fact that all of the eigenvalues $(0, -1)$ are distinct and not positive.

Example 8.4. Decoupling of a Vector Equation Through Diagonalization.

Consider a discrete-time LTI state equation

$$\begin{bmatrix} x_1[n+1] \\ x_2[n+1] \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0.2 & 0.1 \end{bmatrix} \begin{bmatrix} x_1[n] \\ x_2[n] \end{bmatrix} + \begin{bmatrix} 0 \\ 2.2361 \end{bmatrix} u_s[n]$$

$$\text{with } \begin{bmatrix} x_1[0] \\ x_2[0] \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ and } u_s[n] = 1 \ \forall \ n \geq 0 \tag{E8.4.1}$$

In order to diagonalize this equation into a form similar to Eq. (E8.3.4), we use MATLAB to find the eigenvalues/eigenvectors and the modal matrix composed of the eigenvectors and finally, do the similarity transformation.

```
A = [0 1;0.2 0.1]; B = [0; 2.2361]; % Eq.(E8.4.1)
[V,L] = eig(A) % V = modal matrix composed of eigenvectors (E8.4.2)
% L = diagonal matrix with eigenvalues on its diagonal
Ap = V^-1*A*V %diagonalize through similarity transformation (E8.4.3)
  % into a diagonal matrix having the eigenvalues on the diagonal
Bp = V^-1*B % (E8.4.3)
```

Then, we get

$$L = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} = \begin{bmatrix} -0.4 & 0 \\ 0 & 0.5 \end{bmatrix}, \quad V = [\mathbf{v}_1 \ \mathbf{v}_2] = \begin{bmatrix} -0.9285 & -0.8944 \\ 0.3714 & -0.4472 \end{bmatrix} \tag{E8.4.2}$$

$$A_p = V^{-1}AV = \begin{bmatrix} -0.4 & 0 \\ 0 & 0.5 \end{bmatrix} \text{ and } B_p = V^{-1}B = \begin{bmatrix} 2.6759 \\ -2.7778 \end{bmatrix} \tag{E8.4.3}$$

so that we can write the diagonalized state equation as

$$\begin{aligned} \begin{bmatrix} w_1[n+1] \\ w_2[n+1] \end{bmatrix} &= \begin{bmatrix} -0.4 & 0 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} w_1[n] \\ w_2[n] \end{bmatrix} + \begin{bmatrix} 2.6759 \\ -2.7778 \end{bmatrix} u_s[n] \\ &= \begin{bmatrix} -0.4w_1[n] + 2.6759 \\ 0.5w_2[n] - 2.7778 \end{bmatrix} \end{aligned} \tag{E8.4.4}$$

Without the input term on the right-hand side of Eq. (E8.4.1), we would have obtained

$$\begin{bmatrix} w_1[n+1] \\ w_2[n+1] \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} w_1[n] \\ w_2[n] \end{bmatrix} = \begin{bmatrix} \lambda_1^{n+1} & w_1[0] \\ \lambda_2^{n+1} & w_2[0] \end{bmatrix} \quad \text{with } \mathbf{w}[0] = V^{-1}\mathbf{x}[0] \tag{E8.4.5}$$

$$\mathbf{x}[n] = V\mathbf{w}[n] = [\mathbf{v}_1 \ \mathbf{v}_2] \begin{bmatrix} w_1[0]\lambda_1^n \\ w_2[0]\lambda_2^n \end{bmatrix} = w_1[0]\lambda_1^n \mathbf{v}_1 + w_2[0]\lambda_2^n \mathbf{v}_2 \tag{E8.4.6}$$

As time goes by (i.e., as n increases), this solution converges and so the discrete-time system turns out to be stable, thanks to the fact that the magnitude of every eigenvalue (−0.4, 0.5) is less than one.

Remark 8.2. Physical Meaning of Eigenvalues and Eigenvectors

1. As illustrated by the above examples, we can use the modal matrix to decouple a set of differential equations so that they can be solved one by one as a scalar differential equation in terms of a single variable and then put together to make the solution for the original vector differential equation.
2. Through the above examples, we can feel the physical significance of the eigenvalues/eigenvectors of the system matrix A in the state equation on its solution. That is, the state of a linear time-invariant (LTI) system described by an N-dimensional continuous-time (differential) state equation has N modes $\{e^{\lambda_i t}; i = 1, \ldots, N\}$, each of which converges/diverges if the sign of the corresponding eigenvalue is negative/positive and proceeds slowly as

the magnitude of the eigenvalue is close to zero. In the case of a discrete-time LTI system described by an N-dimensional difference state equation, its state has N modes $\{\lambda_i^n; i = 1, \ldots, N\}$, each of which converges/diverges if the magnitude of the corresponding eigenvalue is less/greater than one and proceeds slowly as the magnitude of the eigenvalue is close to one. To summarize, the convergence property of a state $\mathbf{x}$ or the stability of a linear-time invariant (LTI) system is determined by the eigenvalues of the system matrix A. As illustrated by (E8.3.9) and (E8.4.6), the corresponding eigenvector determines the direction in which each mode proceeds in the N-dimensional state space.

8.3 POWER METHOD

In this section, we will introduce the scaled power method, the inverse power method and the shifted inverse power method, to find the eigenvalues of a given matrix.

8.3.1 Scaled Power Method

This method is used to find the eigenvalue of largest magnitude and is summarized in the following box.

SCALED POWER METHOD

Suppose all of the eigenvalues of an $N \times N$ matrix A are distinct with the magnitudes

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_N|$$

Then, the dominant eigenvalue λ_1 with the largest magnitude and its corresponding eigenvector $\mathbf{v}_1$ can be obtained by starting with an initial vector $\mathbf{x}_0$ that has some nonzero component in the direction of $\mathbf{v}_1$ and by repeating the following procedure:

Divide the previous vector $\mathbf{x}_k$ by its largest component (in absolute value) for normalization (scaling) and premultiply the normalized vector by the matrix A.

$$\mathbf{x}_{k+1} = A\frac{\mathbf{x}_k}{\|\mathbf{x}_k\|_\infty} \rightarrow \lambda_1 \mathbf{v}_1 \qquad \text{with } \|\mathbf{x}\|_\infty = \text{Max}\ \{|x_n|\} \tag{8.3.1}$$

Proof. According to Theorem 8.1, the eigenvectors $\{\mathbf{v}_n; n = 1 : N\}$ of an $N \times N$ matrix A whose eigenvalues are distinct are independent and thus can constitute a basis for an N-dimensional linear space. Consequently, any initial vector $\mathbf{x}_0$ can be expressed as a linear combination of the eigenvectors:

$$\mathbf{x}_0 = \alpha_1\mathbf{v}_1 + \alpha_2\mathbf{v}_2 + \cdots + \alpha_N\mathbf{v}_N \tag{8.3.2}$$

Noting that $A\mathbf{v}_n = \lambda_n\mathbf{v}_n$, we premultiply both sides of this equation by A to get

$$\begin{aligned} A\mathbf{x}_0 &= \alpha_1\lambda_1\mathbf{v}_1 + \alpha_2\lambda_2\mathbf{v}_2 + \cdots + \alpha_N\lambda_N\mathbf{v}_N \\ &= \lambda_1\left(\alpha_1\mathbf{v}_1 + \alpha_2\frac{\lambda_2}{\lambda_1}\mathbf{v}_2 + \cdots + \alpha_N\frac{\lambda_N}{\lambda_1}\mathbf{v}_N\right) \end{aligned}$$

and repeat this multiplication over and over again to obtain

$$\begin{aligned} \mathbf{x}_k &= A^k\mathbf{x}_0 \\ &= \lambda_1^k\left\{\alpha_1\mathbf{v}_1 + \alpha_2\left(\frac{\lambda_2}{\lambda_1}\right)^k\mathbf{v}_2 + \cdots + \alpha_N\left(\frac{\lambda_N}{\lambda_1}\right)^k\mathbf{v}_N\right\} \to \lambda_1^k\alpha_1\mathbf{v}_1 \end{aligned} \tag{8.3.3}$$

which will converge to an eigenvector $\mathbf{v}_1$ as long as $\alpha_1 \neq 0$. Since we keep scaling before multiplying at every iteration, the largest component of the limit vector of the sequence generated by Eq. (8.3.1) must be λ_1.

$$\mathbf{x}_{k+1} = A\frac{\mathbf{x}_k}{\|\mathbf{x}_k\|_\infty} \to A\frac{\mathbf{v}_1}{\|\mathbf{v}_1\|_\infty} \overset{(8.1.1)}{=} \lambda_1\frac{\mathbf{v}_1}{\|\mathbf{v}_1\|_\infty} \tag{8.3.4}$$

Note that the scaling prevents the overflow or underflow that would result from $|\lambda_1| > 1$ or $|\lambda_1| < 1$.

Remark 8.3. Convergence of Power Method

1. In the light of Eq. (8.3.3), the convergence speed of the power method depends on how small the magnitude ratio ($|\lambda_2|/|\lambda_1|$) of the second largest eigenvalue λ_2 over the largest eigenvalue λ_1 is.
2. We often use $\mathbf{x}_0 = [\,1 \quad 1 \quad \cdots \quad 1\,]$ as the initial vector. Note that if it has no component in the direction of the eigenvector ($\mathbf{v}_1$) corresponding to the dominant eigenvalue λ_1—that is, $\alpha_1 = \mathbf{x}_0 \bullet \mathbf{v}_1/\|\mathbf{v}_1\|^2 = 0$ in Eq. (8.3.2)—the iteration of the scaled power method leads to the limit showing the second largest magnitude eigenvalue λ_2 and its corresponding eigenvector $\mathbf{v}_2$. But, if there is more than one largest (dominant) eigenvalue of equal magnitude, it does not converge to either of them.

8.3.2 Inverse Power Method

The objective of this method is to find the (uniquely) smallest (magnitude) eigenvalue λ_N by applying the scaled power method to the inverse matrix A^{-1} and taking the inverse of the largest component of the limit. It works only in cases where the matrix A is nonsingular and thus has no zero eigenvalue. Its idea is based on the equation

$$A\mathbf{v} = \lambda\mathbf{v} \rightarrow A^{-1}\mathbf{v} = \lambda^{-1}\mathbf{v} \tag{8.3.5}$$

obtained from multiplying both sides of Eq. (8.1.1) by $\lambda^{-1}A^{-1}$. This implies that the inverse matrix A^{-1} has the eigenvalues that are the reciprocals of the eigenvalues of the original matrix A, still having the same eigenvectors.

$$\lambda_N = \frac{1}{\text{the largest eigenvalue of } A^{-1}} \tag{8.3.6}$$

8.3.3 Shifted Inverse Power Method

In order to develop a method for finding the eigenvalue that is not necessarily of the largest or smallest magnitude, we subtract $s\,\mathbf{v}$ (s: a number that does not happen to equal any eigenvalue) from both sides of Eq. (8.1.1) to write

$$A\mathbf{v} = \lambda\mathbf{v} \rightarrow [A - sI]\mathbf{v} = (\lambda - s)\mathbf{v} \tag{8.3.7}$$

Since this implies that $(\lambda - s)$ is the eigenvalue of $[A - sI]$, we apply the inverse power method for $[A - sI]$ to get its smallest magnitude eigenvalue $(\lambda_k - s)$ with $\min\{|\lambda_i - s|, i = 1 : N\}$ and add s to it to obtain the eigenvalue of the original matrix A which is closest to the number s.

$$\lambda_s = \frac{1}{\text{the largest eigenvalue of } [A - sI]^{-1}} + \mathrm{s} \tag{8.3.8}$$

The prospect of this method is supported by Gerschgorin's disk theorem, which is summarized in the box below. But, this method is not applicable to the matrix that has more than one eigenvalue of the same magnitude.

Theorem 8.2. Gerschgorin's Disk Theorem.

Every eigenvalue of a square matrix A belongs to at least one of the disks (in the complex plane) with center a_{mm} (one of the diagonal elements of A) and radius

$$r_m = \sum_{n \neq m} |a_{mn}| \text{(the sum of all the elements in the row except the diagonal element)}$$

Moreover, each of the disks contains at least one eigenvalue of the matrix A.

The power method introduced in Section 8.3.1 is cast into the routine "`eig_power()`". The MATLAB program "`nm831.m`" uses it to perform the power method, the inverse power method and the shifted inverse power method for finding the eigenvalues of a matrix and compares the results with that of the MATLAB built-in routine "`eig()`" for cross-check.

```
function [lambda,v] = eig_power(A,x,EPS,MaxIter)
% The power method to find the largest eigenvalue (lambda) and
% the corresponding eigenvector (v) of a matrix A.
if nargin < 4, MaxIter = 100; end % maximum number of iterations
if nargin < 3, EPS = 1e-8; end % difference between successive values
N = size(A,2);
if nargin < 2, x = [1:N]; end % the initial vector
x = x(:);
lambda = 0;
for k = 1:MaxIter
  x1 = x;  lambda1 = lambda;
  x = A*x/norm(x,inf); %Eq.(8.3.4)
  [xm,m] = max(abs(x));
  lambda = x(m); % the component with largest magnitude(absolute value)
  if norm(x1 - x) < EPS & abs(lambda1-lambda) < EPS, break; end
end
if k == MaxIter, disp('Warning: you may have to increase MaxIter'); end
v = x/norm(x);
```

```
%nm831
%Apply the power method to find the largest/smallest/medium eigenvalue
clear
A = [2 0 1;0 -2 0;1 0 2];
x = [1 2 3]'; %x = [1 1 1]'; % with different initial vector
EPS = 1e-8; MaxIter = 100;
%the largest eigenvalue and its corresponding eigenvector
[lambda_max,v] = eig_power(A,x,EPS,MaxIter)
%the smallest eigenvalue and its corresponding eigenvector
[lambda,v] = eig_power(A^ - 1,x,EPS,MaxIter);
lambda_min = 1/lambda,  v %Eq.(8.3.6)
%eigenvalue nearest to a number and its corresponding eigenvector
s = -3;  AsI = (A - s*eye(size(A)))^ - 1;
[lambda,v] = eig_power(AsI,x,EPS,MaxIter);
lambda = 1/lambda+s %Eq.(8.3.8)
fprintf('Eigenvalue closest to %4.2f = %8.4f\nwith eigenvector',s,lambda)
v
[V,LAMBDA] = eig(A) %modal matrix composed of eigenvectors
```

8.4 JACOBI METHOD

This method finds us all the eigenvalues of a real symmetric matrix. Its idea is based on the following theorem.

Theorem 8.3. Symmetric Diagonalization Theorem.

All of the eigenvalues of an $N \times N$ symmetric matrix A are of real value and its eigenvectors form an orthonormal basis of an N-dimensional linear space. Consequently, we can make an orthonormal modal matrix V composed of the eigenvectors such that $V^T V = I$; $V^{-1} = V^T$ and use the modal matrix to make the similarity transformation of A, which yields a diagonal matrix having the eigenvalues on its main diagonal:

$$V^T A V = V^{-1} A V = \Lambda \tag{8.4.1}$$

Now, in order to understand the Jacobi method, we define the pq-rotation matrix as

$$R_{pq}(\theta) = \begin{array}{c} p^{\text{th}} \text{ column} \qquad q^{\text{th}} \text{ column} \\ \left[\begin{array}{ccccccccc} 1 & 0 & \cdot & 0 & \cdot & 0 & \cdot & 0 \\ 0 & 1 & \cdot & 0 & \cdot & 0 & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \cos\theta & \cdot & -\sin\theta & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & \sin\theta & \cdot & \cos\theta & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & \cdot & 0 & \cdot & 0 & \cdot & 1 \end{array}\right] \end{array} \begin{array}{l} \\ \\ \\ p^{\text{th}} \text{ row} \\ \\ q^{\text{th}} \text{ row} \\ \\ \\ \end{array} \tag{8.4.2}$$

Since this is an orthonormal matrix whose row/column vectors are orthogonal and normalized

$$R_{pq}^T R_{pq} = I, \qquad R_{pq}^T = R_{pq}^{-1} \tag{8.4.3}$$

premultiplying/postmultiplying a matrix A by R_{pq}^T / R_{pq} makes a similarity transformation

$$A_{(1)} = R_{pq}^T A \ R_{pq} \tag{8.4.4}$$

Noting that the similarity transformation does not change the eigenvalues (Remark 8.1), any matrix resulting from repeating the same operations successively

$$A_{(k+1)} = R_{(k)}^T A_{(k)} R_{(k)} = R_{(k)}^T R_{(k-1)}^T \cdots R^T A R \cdots R_{(k-1)} R_{(k)} \tag{8.4.5}$$

has the same eigenvalues. Moreover, if it is a diagonal matrix, it will have all the eigenvalues on its main diagonal, and the matrix multiplied on the right of the matrix A is the modal matrix V

$$V = R \cdots R_{(k-1)} R_{(k)} \tag{8.4.6}$$

as manifested by matching this equation with Eq. (8.4.1).

```
function [LAMBDA,V,ermsg] = eig_Jacobi(A,EPS,MaxIter)
%Jacobi method finds the eigenvalues/eigenvectors of symmetric matrix A
if nargin < 3, MaxIter = 100; end
if nargin < 2, EPS = 1e-8; end
N = size(A,2);
LAMBDA =[]; V = [];
for m = 1:N
   if norm(A(m:N,m) - A(m,m:N)') > EPS
     error('asymmetric matrix!');
   end
end
V = eye(N);
for k = 1:MaxIter
  for m = 1:N - 1
     [Am(m),Q(m)] = max(abs(A(m,m + 1:N)));
  end
  [Amm,p] = max(Am);  q = p + Q(p);
  if Amm < EPS*sum(abs(diag(LAMBDA))), break; end
  if abs(A(p,p)-A(q,q))<EPS
     s2 = 1;  s = 1/sqrt(2); c = s;
     cc = c*c; ss = s*s;
   else
     t2 = 2*A(p,q)/(A(p,p)- A(q,q));      %Eq.(8.4.9a)
     c2 = 1/sqrt(1 + t2*t2); s2 = t2*c2;  %Eq.(8.4.9b,c)
     c = sqrt((1 + c2)/2); s = s2/2/c;    %Eq.(8.4.9d,e)
     cc = c*c; ss = s*s;
  end
  LAMBDA = A;
  LAMBDA(p,:) = A(p,:)*c + A(q,:)*s;   %Eq.(8.4.7b)
  LAMBDA(:,p) = LAMBDA(p,:)';
  LAMBDA(q,:) = -A(p,:)*s + A(q,:)*c;   %Eq.(8.4.7c)
  LAMBDA(:,q) = LAMBDA(q,:)';
  LAMBDA(p,q) = 0;  LAMBDA(q,p) = 0;   %Eq.(8.4.7a)
  LAMBDA(p,p) = A(p,p)*cc +A(q,q)*ss + A(p,q)*s2; %Eq.(8.4.7d)
  LAMBDA(q,q) = A(p,p)*ss +A(q,q)*cc - A(p,q)*s2; %Eq.(8.4.7e)
  A = LAMBDA;
  V(:,[p q]) = V(:,[p q])*[c -s;s c];
end
LAMBDA = diag(diag(LAMBDA)); %for purification
```

```
%nm841 applies the Jacobi method
% to find all the eigenvalues/eigenvectors of a symmetric matrix A.
clear
A = [2 0 1;0 -2 0;1 0 2];
EPS = 1e-8;  MaxIter =100;
[L,V] = eig_Jacobi(A,EPS,MaxIter)
disp('Using eig()')
[V,LAMBDA] = eig(A) %modal matrix composed of eigenvectors
```

What is left for us to think about is how to make this matrix (8.4.5) diagonal. Noting that the similarity transformation (8.4.4) changes only the pth rows/columns and the qth rows/columns as

$$\begin{aligned} v_{pq} = v_{qp} &= a_{qp}(c^2 - s^2) + (a_{qq} - a_{pp})sc \\ &= a_{qp}\cos 2\theta + \frac{1}{2}(a_{qq} - a_{pp})\sin 2\theta \end{aligned} \tag{8.4.7a}$$

$$v_{pn} = v_{np} = a_{pn}c + a_{qn}s \quad \text{for the } p^{\text{th}} \text{ row/column with } n \neq p, q \tag{8.4.7b}$$

$$v_{qn} = v_{np} = -a_{pn}s + a_{qn}c \quad \text{for the } q^{\text{th}} \text{ row/column with } n \neq p, q \tag{8.4.7c}$$

$$v_{pp} = a_{pp}c^2 + a_{qq}s^2 + 2a_{pq}sc = a_{pp}c^2 + a_{qq}s^2 + a_{pq}\sin 2\theta \tag{8.4.7d}$$

$$v_{qq} = a_{pp}s^2 + a_{qq}c^2 - 2a_{pq}sc = a_{pp}s^2 + a_{qq}c^2 - a_{pq}\sin 2\theta \tag{8.4.7e}$$

$$(c = \cos\theta, s = \sin\theta)$$

we make the (p, q) element v_{pq} and the (q, p) element v_{qp} zero

$$v_{pq} = v_{qp} = 0 \tag{8.4.8}$$

by choosing the angle θ of the rotation matrix $R_{pq}(\theta)$ in such a way that

$$\tan 2\theta = \frac{\sin 2\theta}{\cos 2\theta} = \frac{2a_{pq}}{a_{pp} - a_{qq}}, \qquad \cos 2\theta = \frac{1}{\sec 2\theta} = \frac{1}{\sqrt{1 + \tan^2 2\theta}},$$
$$\sin 2\theta = \tan 2\theta \cos 2\theta \tag{8.4.9}$$
$$\cos\theta = \sqrt{\cos^2\theta} = \sqrt{(1 + \cos 2\theta)/2}, \qquad \sin\theta = \frac{\sin 2\theta}{2\cos\theta}$$

and computing the other associated elements according to Eqs. (8.4.7b–e).

There are a couple of things to note. First, in order to make the matrix closer to a diagonal one at each iteration, we should identify the row number and the column number of the largest off-diagonal element as p and q, respectively, and zero-out the (p, q) element. Second, we can hope that the magnitudes of the other elements in the pth,qth row/column affected by this transformation process don't get larger, since Eqs. (8.4.7b) and (8.4.7c) implies

$$v_{pn}^2 + v_{qn}^2 = (a_{pn}c + a_{qn}s)^2 + (-a_{pn}s + a_{qn}c)^2 = a_{pn}^2 + a_{qn}^2 \tag{8.4.10}$$

This so-called Jacobi method is cast into the routine "`eig_Jacobi()`". The MATLAB program "`nm841.m`" uses it to find the eigenvalues/eigenvectors of a matrix and compares the result with that of using the MATLAB built-in routine "`eig()`" for cross-check. The result we may expect is as follows. Interested readers are welcome to run the program "`nm841.m`".

$$A = \begin{bmatrix} 2 & 0 & 1 \\ 0 & -2 & 0 \\ 1 & 0 & 2 \end{bmatrix} \rightarrow R_{13}^T A R_{13} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \Lambda$$

$$\text{with } R_{13} = \begin{bmatrix} 1/\sqrt{2} & 0 & -1/\sqrt{2} \\ 0 & 1 & 0 \\ 1/\sqrt{2} & 0 & 1/\sqrt{2} \end{bmatrix} = V$$

8.5 PHYSICAL MEANING OF EIGENVALUES/EIGENVECTORS

According to Theorem 8.3 (Symmetric Diagonalization Theorem), introduced in the previous section, the eigenvectors $\{\mathbf{v}_n, n = 1 : N\}$ of an $N \times N$ symmetric matrix A constitute an orthonormal basis for an N-dimensional linear space.

$$V^T V = I, \qquad \mathbf{v}_m^T \mathbf{v}_n = \delta_{mn} = \begin{cases} 1 & \text{for } m = n \\ 0 & \text{for } m \neq n \end{cases} \tag{8.5.1}$$

Consequently, any N-dimensional vector $\mathbf{x}$ can be expressed as a linear combination of these eigenvectors.

$$\mathbf{x} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \cdots + \alpha_N \mathbf{v}_N = \sum_{n=1}^{N} \alpha_n \mathbf{v}_n \tag{8.5.2}$$

Thus, the eigenvectors are called the principal axes of matrix A, and the squared norm of a vector is the sum of the squares of the components (α_n's) along the principal axis.

$$\|\mathbf{x}\|^2 = \mathbf{x}^T \mathbf{x} = \left(\sum_{m=1}^{N} \alpha_m \mathbf{v}_m \right)^T \left(\sum_{n=1}^{N} \alpha_n \mathbf{v}_n \right) = \sum_{m=1}^{N} \sum_{n=1}^{N} \alpha_m \alpha_n \mathbf{v}_m^T \mathbf{v}_n = \sum_{n=1}^{N} \alpha_n^2 \tag{8.5.3}$$

Premultiplying Eq. (8.5.2) by the matrix A and using Eq. (8.1.1) yields

$$A\mathbf{x} = \lambda_1 \alpha_1 \mathbf{v}_1 + \lambda_2 \alpha_2 \mathbf{v}_2 + \cdots + \lambda_N \alpha_N \mathbf{v}_N = \sum_{n=1}^{N} \lambda_n \alpha_n \mathbf{v}_n \tag{8.5.4}$$

This shows that premultiplying a vector $\mathbf{x}$ by matrix A has the same effect as multiplying each principal component α_n of $\mathbf{x}$ along the direction of eigenvector $\mathbf{v}_n$ by the associated eigenvalue λ_n. Therefore, the solution of a homogeneous discrete-time state equation

$$\mathbf{x}(k+1) = A\mathbf{x}(k) \qquad \text{with } \mathbf{x}(0) = \sum_{n=1}^{N} \alpha_n \mathbf{v}_n \tag{8.5.5}$$

can be written as

$$\mathbf{x}(k) = \sum_{n=1}^{N} \lambda_n^k \alpha_n \mathbf{v}_n \tag{8.5.6}$$

which was illustrated by Eq. (E8.4.6) in Example 8.4. On the other hand, as illustrated by (E8.3.9) in Example 8.3(b), the solution of a homogeneous continuous-time state equation

$$\mathbf{x}'(t) = A\mathbf{x}(t) \qquad \text{with } \mathbf{x}(0) = \sum_{n=1}^{N} \alpha_n \mathbf{v}_n \tag{8.5.7}$$

can be written as

$$\mathbf{x}(t) = \sum_{n=1}^{N} e^{\lambda_n t} \alpha_n \mathbf{v}_n \tag{8.5.8}$$

Equations (8.5.6) and (8.5.8) imply that the eigenvalues of the system matrix characterize the principal modes of the system described by the state equations. That is, the eigenvalues determine not only whether the system is stable or not—that is, whether the system state converges to an equilibrium state or diverges—but also how fast the system state proceeds along the direction of each eigenvector. More specifically, in the case of a discrete-time system, the absolute values of all the eigenvalues must be less than one for stability and the smaller the absolute value of an eigenvalue (less than one) is, the faster the corresponding mode converges. In the case of a continuous-time system, the real parts of all the eigenvalues must be negative for stability and the smaller a negative eigenvalue is, the faster the corresponding mode converges. The difference among the eigenvalues determines how stiff the system is (see Section 6.5.4). This meaning of eigenvalues/eigenvectors is very important in dynamic systems.

Now, in order to figure out the meaning of eigenvalues/eigenvectors in static systems, we define the mean vector and the covariance matrix of the vectors $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(K)}\}$ representing K points in a two-dimensional space called the x_1x_2 plane as

$$\mathbf{m}_x = \frac{1}{K}\sum_{k=1}^{K} \mathbf{x}^{(k)}, \qquad C_x = \frac{1}{K}\sum_{k=1}^{K} [\mathbf{x}^{(k)} - \mathbf{m}_x][\mathbf{x}^{(k)} - \mathbf{m}_x]^T \tag{8.5.9}$$

where the mean vector represents the center of the points and the covariance matrix describes how dispersedly the points are distributed. Let us think about the geometrical meaning of diagonalizing the covariance matrix C_x. As a simple example, suppose we have four points

$$\mathbf{x}^{(1)} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad \mathbf{x}^{(2)} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad \mathbf{x}^{(3)} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad \mathbf{x}^{(4)} = \begin{bmatrix} 3 \\ 2 \end{bmatrix} \tag{8.5.10}$$

for which the mean vector $\mathbf{m}_x$, the covariance matrix C_x, and its modal matrix are

$$\mathbf{m}_x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad C_x = \begin{bmatrix} 2.5 & 2 \\ 2 & 2.5 \end{bmatrix}, \quad V = [\mathbf{v}_1 \quad \mathbf{v}_2] = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \tag{8.5.11}$$

Then, we can diagonalize the covariance matrix as

$$\begin{aligned} V^T C_x V &= \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} 2.5 & 2 \\ 2 & 2.5 \end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.5 & 0 \\ 0 & 4.5 \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} = \Lambda \end{aligned} \tag{8.5.12}$$

which has the eigenvalues on its main diagonal. On the other hand, if we transform the four point vectors by using the modal matrix as

$$\mathbf{y} = V^T(\mathbf{x} - \mathbf{m}_x) \tag{8.5.13}$$

then the new four point vectors are

$$\mathbf{y}^{(1)} = \begin{bmatrix} 1/\sqrt{2} \\ -3/\sqrt{2} \end{bmatrix},\ \mathbf{y}^{(2)} = \begin{bmatrix} -1/\sqrt{2} \\ -3/\sqrt{2} \end{bmatrix},\ \mathbf{y}^{(3)} = \begin{bmatrix} -1/\sqrt{2} \\ 3/\sqrt{2} \end{bmatrix},\ \mathbf{y}^{(4)} = \begin{bmatrix} 1/\sqrt{2} \\ 3/\sqrt{2} \end{bmatrix} \tag{8.5.14}$$

for which the mean vector $\mathbf{m}_x$ and the covariance matrix C_x are

$$\mathbf{m}_y = V^T(\mathbf{m}_x - \mathbf{m}_x) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \qquad C_y = V^T C_x V = \begin{bmatrix} 0.5 & 0 \\ 0 & 4.5 \end{bmatrix} = \Lambda \tag{8.5.15}$$

The original four points and the new points corresponding to them are depicted in Fig. 8.1, which shows that the eigenvectors of the covariance matrix for a set of point vectors represents the principal axes of the distribution and its eigenvalues are related with the lengths of the distribution along the principal axes. The difference among the eigenvalues determines how oblong the overall shape of the distribution is.

Before closing this section, we may think about the meaning of the determinant of a matrix composed of two two-dimensional vectors and three three-dimensional vectors.

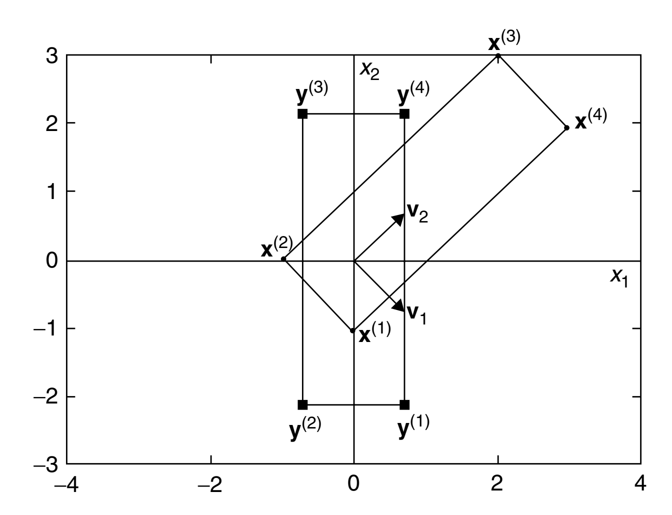


Figure 8.1 Eigenvalues/eigenvectors of a covariance matrix.

First, let us consider a 2×2 matrix composed of two two-dimensional vectors $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$.

$$X = [\,\mathbf{x}^{(1)} \quad \mathbf{x}^{(2)}\,] = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \tag{8.5.16}$$

Conclusively, the absolute value of the determinant of this matrix

$$\det(X) = |X| = x_{11}x_{22} - x_{12}x_{21} \tag{8.5.17}$$

equals the area of the parallelogram having the two vectors as its two neighboring sides. In order to certify this fact, let us make a clockwise rotation of the two vectors by the phase angle of $\mathbf{x}^{(1)}$

$$-\theta_1 = -\tan^{-1}\left(\frac{x_{21}}{x_{11}}\right) \tag{8.5.18}$$

so that the new vector $\mathbf{y}^{(1)}$ corresponding to $\mathbf{x}^{(1)}$ becomes aligned with the x_1-axis (see Fig. 8.2). For this purpose, we multiply our matrix X by the rotation matrix defined by Eq. (8.4.2)

$$R(-\theta_1) = \begin{bmatrix} \cos\theta_1 & -\sin(-\theta_1) \\ \sin(-\theta_1) & \cos\theta_1 \end{bmatrix} = \frac{1}{\sqrt{x_{11}^2 + x_{21}^2}} \begin{bmatrix} x_{11} & x_{21} \\ -x_{21} & x_{11} \end{bmatrix} \tag{8.5.19}$$

to get

$$Y = R(-\theta_1)X = \frac{1}{\sqrt{x_{11}^2 + x_{21}^2}} \begin{bmatrix} x_{11} & x_{21} \\ -x_{21} & x_{11} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \tag{8.5.20a}$$

$$[\,\mathbf{y}^{(1)} \quad \mathbf{y}^{(2)}\,] = \frac{1}{\sqrt{x_{11}^2 + x_{21}^2}} \begin{bmatrix} x_{11}^2 + x_{21}^2 & x_{11}x_{12} + x_{21}x_{22} \\ 0 & -x_{12}x_{21} + x_{11}x_{22} \end{bmatrix} \tag{8.5.20b}$$

The parallelograms having the original vectors and the new vectors as their two neighboring sides are depicted in Fig. 8.2, where the areas of the parallelograms turn out to be equal to the absolute values of the determinants of the matrices X and Y as follows:

$$\begin{aligned} &\text{Area of the parallelograms} \\ &= \text{Length of the bottom side} \times \text{Height of the parallelogram} \\ &= (x_1 \text{ component of } \mathbf{y}^{(1)}) \times (x_2 \text{ component of } \mathbf{y}^{(2)}) = y_{11}y_{22} = \det(Y) \\ &= \frac{x_{11}^2 + x_{21}^2}{\sqrt{x_{11}^2 + x_{21}^2}} \times \frac{-x_{12}x_{21} + x_{11}x_{22}}{\sqrt{x_{11}^2 + x_{21}^2}} \equiv \det(X) \end{aligned} \tag{8.5.21}$$

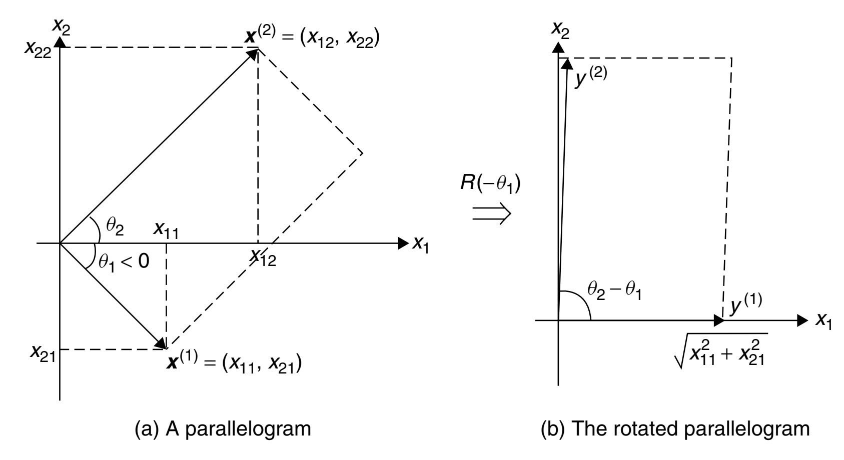


Figure 8.2 Geometrical meaning of a determinant.

On extension of this result into a three-dimensional situation, the absolute value of the determinant of a 3×3 matrix composed of three three-dimensional vectors $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}$, and $\mathbf{x}^{(3)}$ equals the volume of the parallelepiped having the three vectors as its three edges.

$$\det(X) = |X| = |\,\mathbf{x}^{(1)} \quad \mathbf{x}^{(2)} \quad \mathbf{x}^{(3)}\,| = \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \equiv \mathbf{x}^{(1)} \times \mathbf{x}^{(2)} \cdot \mathbf{x}^{(3)} \tag{8.5.22}$$

8.6 EIGENVALUE EQUATIONS

In this section, we consider a system of ordinary differential equations that can be formulated as an eigenvalue problem.

For the undamped mass–spring system depicted in Fig. 8.3, the displacements $x_1(t)$ and $x_2(t)$ of the two masses m_1 and m_2 are described by the following system of differential equations:

$$\begin{bmatrix} x_1''(t) \\ x_2''(t) \end{bmatrix} = -\begin{bmatrix} (k_1 + k_2)/m_1 & -k_2/m_1 \\ -k_2/m_2 & k_2/m_2 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}$$

$$\text{with } \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} \text{ and } \begin{bmatrix} x_1'(0) \\ x_2'(0) \end{bmatrix}$$

$$\mathbf{x}''(t) = -A\mathbf{x}(t) \qquad \text{with } \mathbf{x}(0) \text{ and } \mathbf{x}'(0) \tag{8.6.1}$$

Let the eigenpairs (eigenvalue–eigenvectors) of the matrix A be $(\lambda_n = \omega_n^2, \mathbf{v}_n)$ with

$$A\mathbf{v}_n = \omega_n^2 \mathbf{v}_n \tag{8.6.2}$$

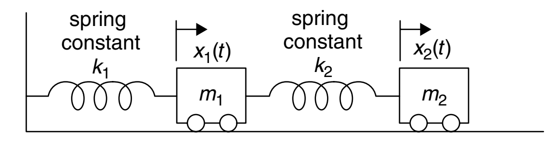


Figure 8.3 An undamped mass–spring system.

Noting that the solution of Eq. (8.5.7) can be written as Eq. (8.5.8) in terms of the eigenvectors of the system matrix, we write the solution of Eq. (8.6.1) as

$$\mathbf{x}(t) = \sum_{n=1}^{2} w_n(t)\mathbf{v}_n = [\,\mathbf{v}_1 \quad \mathbf{v}_2\,]\begin{bmatrix} w_1(t) \\ w_2(t) \end{bmatrix} = V\mathbf{w}(t) \tag{8.6.3}$$

and substitute this into Eq. (8.6.1) to have

$$\sum_{n=1}^{2} w_n''(t)\mathbf{v}_n = -A\sum_{n=1}^{2} w_n(t)\mathbf{v}_n \overset{(8.6.2)}{=} -\sum_{n=1}^{2} w_n(t)\omega_n^2\mathbf{v}_n \tag{8.6.4}$$

$$w_n''(t) = -\omega_n^2 w_n(t) \qquad \text{for } n = 1, 2 \tag{8.6.5}$$

The solution of this equation is

$$w_n(t) = w_n(0)\cos(\omega_n t) + \frac{w_n'(0)}{\omega_n}\sin(\omega_n t) \qquad \text{with } \omega_n = \sqrt{\lambda_n} \text{ for } n = 1, 2 \tag{8.6.6}$$

where the initial value of $\mathbf{w}(t) = [w_1(t)\ w_2(t)]^T$ can be obtained via Eq. (8.6.3) from that of $\mathbf{x}(t)$ as

$$\mathbf{w}(0) \overset{(8.6.3)}{=} V^{-1}\mathbf{x}(0) \overset{(8.4.1)}{=} V^T\mathbf{x}(0), \qquad \mathbf{w}'(0) = V^T\mathbf{x}'(0) \tag{8.6.7}$$

Finally, we substitute Eq. (8.6.6) into Eq. (8.6.3) to obtain the solution of Eq. (8.6.1).

PROBLEMS

8.1 Symmetric Tridiagonal Toeplitz Matrix

Consider the following $N \times N$ symmetric tridiagonal Toeplitz matrix as

$$\begin{bmatrix} a & b & 0 & \cdot\cdot & 0 & 0 \\ b & a & b & \cdot\cdot & 0 & 0 \\ 0 & b & a & \cdot\cdot & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot\cdot & a & b \\ 0 & 0 & 0 & \cdot\cdot & b & a \end{bmatrix} \tag{P8.1.1}$$

(a) Verify that the eigenvalues and eigenvectors of this matrix are as follows, with $N = 3$ for convenience.

$$\lambda_n = a + 2b\cos\left(\frac{n\pi}{N+1}\right) \qquad \text{for } n = 1 \text{ to } N \tag{P8.1.2}$$

$$\mathbf{v}_n = \sqrt{\frac{2}{N+1}}\left[\sin\left(\frac{n\pi}{N+1}\right)\ \sin\left(\frac{2n\pi}{N+1}\right)\cdots\sin\left(\frac{Nn\pi}{N+1}\right)\right]^T \tag{P8.1.3}$$

(b) Letting $N = 3, a = 2$, and $b = 1$, find the eigenvalues/eigenvectors of the above matrix by using (P8.1.2,3) and by using the MATLAB routine "`eig_Jacobi()`" or "`eig()`" for cross-check.

8.2 Circulant Matrix

Consider the following $N \times N$ circulant matrix as

$$\begin{bmatrix} h(0) & h(N-1) & h(N-2) & \cdot\cdot & h(1) \\ h(1) & h(0) & h(N-1) & \cdot\cdot & h(2) \\ h(2) & h(1) & h(0) & \cdot\cdot & h(3) \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot \\ h(N-1) & h(N-2) & h(N-3) & \cdot\cdot & h(0) \end{bmatrix} \tag{P8.2.1}$$

(a) Vertify that the eigenvalues and eigenvectors of this matrix are as follows, with $N = 4$ for convenience.

$$\lambda_n = h(0) + h(N-1)e^{j2\pi n/N} + h(N-2)e^{j2\pi 2n/N} \tag{P8.2.2}$$
$$+ \cdots + h(1)e^{j2\pi(N-1)n/N}$$

$$\mathbf{v}_n = [1\ e^{j2\pi n/N}\ e^{j2\pi 2n/N}\cdots\ e^{j2\pi(N-1)n/N}]^T \tag{P8.2.3}$$

$$\text{for } n = 0 \text{ to } N-1$$

(b) Letting $N = 4, h(0) = 2, h(3) = h(1) = 1$, and $h(2) = 0$, find the eigenvalues/eigenvectors of the above matrix by using (P8.2.2,3) and by using the MATLAB routine "`eig_Jacobi()`" or "`eig()`". Do they agree? Do they satisfy Eq. (8.1.1)?

8.3 Solving a Vector Differential Equation by Decoupling: Diagonalization. Consider the following two-dimensional vector differential equation (state equation) as

$$\begin{bmatrix} x_1{}'(t) \\ x_2{}'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_s(t) \tag{P8.3.1}$$

$$\text{with } \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } u_s(t) = 1\ \forall\ t \ge 0$$

which was solved by using Laplace transform in Problem P6.1. In this problem, we solve it again by the decoupling method through diagonalization of the system matrix.

(a) Show that the eigenvalues and eigenvectors of the system matrix are as follows.

$$\lambda_1 = -1, \quad \lambda_2 = -2; \qquad \mathbf{v}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad \mathbf{v}_2 = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \tag{P8.3.2}$$

(b) Show that the diagonalization of the above vector differential equation using the modal matrix $V = [\,\mathbf{v}_1 \quad \mathbf{v}_2\,]$ yields the following equation:

$$\begin{bmatrix} w_1{}'(t) \\ w_2{}'(t) \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} w_1(t) \\ w_2(t) \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \end{bmatrix} u_s(t) \tag{P8.3.3}$$

$$= \begin{bmatrix} -w_1(t) + u_s(t) \\ -2w_2(t) - u_s(t) \end{bmatrix} \quad \text{with} \quad \begin{bmatrix} w_1(0) \\ w_2(0) \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

(c) Show that these equations can be solved individually by using Laplace transform technique to yield the following solution, which is the same as Eq. (P6.1.2) obtained in Problem P6.1(a).

$$W_1(s) = \frac{1}{s} + \frac{1}{s+1}, \qquad w_1(t) = (1 + e^{-t})u_s(t) \tag{P8.3.4a}$$

$$W_2(s) = \frac{-1/2}{s} - \frac{1/2}{s+2}, \qquad w_2(t) = -\frac{1}{2}(1 + e^{-2t})u_s(t) \tag{P8.3.4b}$$

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} 1/2 + e^{-t} - (1/2)e^{-2t} \\ -e^{-t} + e^{-2t} \end{bmatrix} u_s(t) \tag{P8.3.5}$$

8.4 Householder Method and QR Factorization

This method can zero-out several elements in a column vector at each iteration and make any $N \times N$ matrix a (lower) triangular matrix in $(N-1)$ iterations.

(a) Householder Reflection (Fig. P8.4)
Show that the transformation matrix by which we can multiply a vector $\mathbf{x}$ to generate another vector $\mathbf{y}$ having the same norm is

$$H = [I - 2\mathbf{w}\mathbf{w}^T]$$

$$\text{with } \mathbf{w} = \frac{\mathbf{x} - \mathbf{y}}{\|\mathbf{x} - \mathbf{y}\|_2} = \frac{1}{c}(\mathbf{x} - \mathbf{y}), \ c = \|\mathbf{x} - \mathbf{y}\|_2, \ \|\mathbf{x}\| = \|\mathbf{y}\| \tag{P8.4.1}$$

and that this is an orthonormal symmetric matrix such that $H^T H = HH = I$; $H^{-1} = H$. Note the following facts.

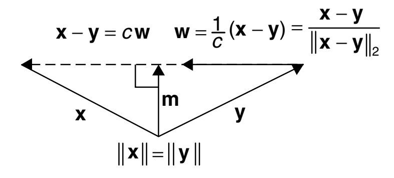


Figure P8.4 Householder reflection.

(i)
$$\mathbf{y} = \mathbf{x} - (\mathbf{x} - \mathbf{y}) \overset{\text{(P8.4.1)}}{=} \mathbf{x} - c\mathbf{w} \tag{P8.4.2a}$$

(ii)
$$\mathbf{w}^T\mathbf{w} \overset{\text{(P8.4.1)}}{=} 1 \text{ and } \|\mathbf{x}\| = \|\mathbf{y}\| \tag{P8.4.2b}$$

(iii)
$$\mathbf{m} = (\mathbf{x} + \mathbf{y})/2 = \mathbf{x} - (c/2)\mathbf{w} \tag{P8.4.2c}$$

(iv) The mean vector **m** of **x** and **y** is orthogonal to the difference vector $\mathbf{w} = (\mathbf{x} - \mathbf{y})/c$.
Thus we have

$$\mathbf{w}^T(\mathbf{x} - (c/2)\mathbf{w}) = 0; \qquad \mathbf{w}^T\mathbf{x} - (c/2)\mathbf{w}^T\mathbf{w} = \mathbf{w}^T\mathbf{x} - (c/2) = 0 \tag{P8.4.3}$$

This gives an expression for $c = \|\mathbf{x} - \mathbf{y}\|_2$ as

$$c = \|\mathbf{x} - \mathbf{y}\|_2 = 2\mathbf{w}^T\mathbf{x} \tag{P8.4.4}$$

We can substitute this into (P8.4.2a) to get the desired result.

$$\mathbf{y} = \mathbf{x} - c\mathbf{w} = \mathbf{x} - 2\mathbf{w}\mathbf{w}^T\mathbf{x} = [I - 2\mathbf{w}\mathbf{w}^T]\mathbf{x} \equiv H\mathbf{x} \tag{P8.4.5}$$

On the other hand, the Householder transform matrix is an orthogonal matrix, since

$$\begin{aligned} H^T H = HH &= [I - 2\mathbf{w}\mathbf{w}^T][I - 2\mathbf{w}\mathbf{w}^T] \\ &= I - 4\mathbf{w}\mathbf{w}^T + 4\mathbf{w}\mathbf{w}^T\mathbf{w}\mathbf{w}^T \\ &= I - 4\mathbf{w}\mathbf{w}^T + 4\mathbf{w}\mathbf{w}^T = I \end{aligned} \tag{P8.4.6}$$

(b) Householder Transform
In order to show that the Householder matrix can be used to zero-out some part of a vector, let us find the kth Householder matrix H_k transforming any vector

$$\mathbf{x} = \begin{bmatrix} x_1 & \cdots & x_{k-1} & x_k & x_{k+1} & \cdots & x_N \end{bmatrix} \tag{P8.4.7}$$

into

$$\mathbf{y} = \begin{bmatrix} x_1 & \cdots & x_{k-1} & -g_k & 0 & \cdots & 0 \end{bmatrix} \tag{P8.4.8}$$

where g_k is fixed in such a way that the norms of these two vectors are the same:

$$g_k = \sqrt{\sum_{n=k}^{N} x_n^2} \tag{P8.4.9}$$

First, we find the difference vector of unit norm as

$$\begin{aligned} \mathbf{w}_k &= \frac{1}{c}(\mathbf{x} - \mathbf{y}) \\ &= \frac{1}{c}\begin{bmatrix} 0 & \cdots & 0 & x_k + g_k & x_{k+1} & \cdots & x_N \end{bmatrix} \end{aligned} \tag{P8.4.10}$$

with

$$c = \|\mathbf{x} - \mathbf{y}\|_2 = \sqrt{(x_k + g_k)^2 + x_{k+1}^2 + \cdots + x_N^2} \tag{P8.4.11}$$

Then, one more thing we should do is to substitute this difference vector into Eq. (P8.4.1).

$$H_k = [I - 2\mathbf{w}_k\mathbf{w}_k^T] \tag{P8.4.12}$$

Complete the following routine "`Householder()`" by permuting the statements and try it with $k = 1$, 2, 3, and 4 for a four-dimensional vector generated by the MATLAB command `rand(5,1)` to check if it works fine.

```
>> x = rand(5,1), for k = 1:4, householder(x,k)*x, end
```

```
function H = Householder(x,k)
%Householder transform to zero out tail part starting from k + 1
H = eye(N) - 2*w*w'; %Householder matrix
N = length(x);
w = zeros(N,1);
w(k) =(x(k) + g)/c; w(k + 1:N) = x(k + 1:N)/c; %Eq.(P8.4.10)
tmp = sum(x(k + 1:N).^ 2);
c = sqrt((x(k) + g)^2 + tmp); %Eq.(P8.4.11)
g = sqrt(x(k)^2 + tmp); %Eq.(P8.4.9)
```

(c) QR Factorization Using Householder Transform
We can use Householder transform to zero out the part under the main diagonal of each column of an $N \times N$ matrix A successively and then make it a lower triangular matrix R in $(N - 1)$ iterations. The necessary operations are collectively written as

$$H_{N-1}H_{N-2}\cdots H_1 A = R \tag{P8.4.13}$$

which implies that

$$A = [H_{N-1}H_{N-2}\cdots H_1]^{-1}R = H_1^{-1}\cdots H_{N-2}^{-1}H_{N-1}^{-1}R$$
$$= H_1\cdots H_{N-2}H_{N-1}R = QR \tag{P8.4.14}$$

where the product of all the Householder matrices

$$Q = H_1\cdots H_{N-2}H_{N-1} \tag{P8.4.15}$$

turns out to be not only symmetric, but also orthogonal like each H_k:

$$Q^TQ = [H_1\cdots H_{N-2}H_{N-1}]^T H_1\cdots H_{N-2}H_{N-1}$$
$$= H_{N-1}^T H_{N-2}^T\cdots H_1^T H_1\cdots H_{N-2}H_{N-1} = I$$

This suggests a QR factorization method that is cast into the following routine "`qr_my()`". You can try it for a nonsingular 3×3 matrix generated by the MATLAB command `rand(3)` and compare the result with that of the MATLAB built-in routine "`qr()`".

```
function [Q,R] = qr_my(A)
%QR factorization
N = size(A,1); R = A; Q = eye(N);
for k = 1:N - 1
   H = Householder(R(:,k),k);
   R = H*R; %Eq.(P8.4.13)
   Q = Q*H; %Eq.(P8.4.15)
end
```

8.5 Hessenberg Form Using Householder Transform

```
function [Hs,HH] = Hessenberg(A)
%Transform into an almost upper triangular matrix
% having only zeros below lower subdiagonal
N = size(A,1); Hs = A; HH = eye(N); %HH*A*HH' = Hs
for k = 1:N - 2
   H = Householder(Hs(:,k),    );
   Hs = H*Hs*H; HH = H*HH;
end
```

We can make use of Householder transform (introduced in Problem 8.4) to zero-out the elements below the lower subdiagonal of a matrix so that it becomes an upper Hessenberg form which is almost upper-triangular matrix. Complete the above routine "`Hessenberg()`" by filling in the second input argument of the routine "`Householder()`" and try it for a 5×5 matrix generated by the MATLAB command `rand(5)` to check if it works.

8.6 QR Factorization of Hessenberg Form Using the Givens Rotation

We can make use of the Givens rotation to get the QR factorization of Hessenberg form by the procedure implemented in the following routine "qr_Hessenberg()", where each element on the lower subdiagonal is zeroed out at each iteration. Generate a 4×4 random matrix A by the MATLAB command rand(4), transform it into a Hessenberg form Hs by using the routine "Hessenberg()" and try this routine "qr_Hessenberg()" for the matrix of Hessenberg form. Check the validity by seeing if norm(Hs-Q*R) $\approx$ 0 or not.

```
function [Q,R] = qr_Hessenberg(Hs)
%QR factorization of Hessenberg form by Givens rotation
N = size(Hs,1);
Q = eye(N); R = Hs;
for k = 1:N - 1
   x = R(k,k); y = R(k+1,k); r = sqrt(x*x + y*y);
   c = x/r; s = -y/r;
   R0 = R; Q0 = Q;
   R(k,:) = c*R0(k,:) - s*R0(k + 1,:);
   R(k + 1,:) = s*R0(k,:) + c*R0(k + 1,:);
   Q(:,k) = c*Q0(:,k) - s*Q0(:,k + 1);
   Q(:,k + 1) = s*Q0(:,k) + c*Q0(:,k + 1);
end
```

8.7 Diagonalization by Using QR Factorization to Find Eigenvalues

You will see that a real symmetric matrix A can be diagonalized into a diagonal matrix having the eigenvalues on its diagonal if we repeat the similarity transformation by using the orthogonal matrix Q obtained from the QR factorization. For this purpose, take the following steps.

```
function [eigs,A] = eig_QR(A,kmax)
%Find eigenvalues by using QR factorization
if nargin < 2, kmax = 200; end
for k = 1:kmax
   [Q,R] = qr(A); %A = Q*R; R =Q'*A =Q^-1*A
   A = R*Q; %A = Q^ - 1*A*Q
end
eigs = diag(A);
```

```
function [eigs,A] = eig_QR_Hs(A,kmax)
%Find eigenvalues by using QR factorization via Hesenberg
if nargin < 2, kmax = 200; end
Hs = hessenberg(A);
for k = 1:kmax
   [Q,R] = qr_hessenberg(Hs); %Hs = Q*R; R = Q'*Hs = Q^ - 1*Hs
   Hs = R*Q; %Hs = Q^ - 1*Hs*Q
end
eigs = diag(Hs);
```

(a) Make the above routine "`eig_QR()`" that uses the MATLAB built-in routine "`qr()`" and then apply it to a 4 × 4 random symmetric matrix `A` generated by the following MATLAB statements.

```
>> A = rand(4); A = A + A';
```

(b) Make the above routine "`eig_QR_Hs()`" that transforms a given matrix into a Hessenberg form by using the routine "`Hessenberg()`" (appeared in Problem 8.5) and then repetitively makes the QR factorization by using the routine "`qr_Hessenberg()`" (appeared in Problem 8.6) and the similarity transformation by the orthogonal matrix Q until the matrix becomes diagonal. Apply it to the 4 × 4 random symmetric matrix `A` generated in (a) and compare the result with those obtained in (a) and by using the MATLAB built-in routine "`eig()`" for cross-check.

8.8 Differential/Difference Equation, State Equation, and Eigenvalue

As mentioned in Section 6.5.3, a high-order scalar differential equation such as

$$x^{(3)}(t) + a_2 x^{(2)}(t) + a_1 x'(t) + a_0 x(t) = u(t) \tag{P8.8.1}$$

can be transformed into a first-order vector differential equation, called a state equation, as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \\ x_3'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -a_0 & -a_1 & -a_2 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u(t) \tag{P8.8.2a}$$

$$x(t) = [1 \quad 0 \quad 0] \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} \tag{P8.8.2b}$$

The characteristic equation of the differential equation (P8.8.1) is

$$s^3 + a_2 s^2 + a_1 s + a_0 = 0 \tag{P8.8.3}$$

and its roots are called the characteristic roots.

(a) What is the relationship between these characteristic roots and the eigenvalues of the system matrix A of the above state equation (P8.8.2)? To answer this question, write the equation $|\lambda I - A| = 0$ to solve for the eigenvalues of A, and show that it is equivalent to Eq. (P8.8.3). To extend your experience or just for practice, you can try the symbolic computation of MATLAB by running the following program "`nm8p08a.m`".

```
%nm8p08a
syms a0 a1 a2 s
A =[0 1 0;0 0 1;-a0 -a1 -a2]; %(P8.8.2a)
det(s*eye(size(A))- A) %characteristic polynomial
ch_eq = poly(A) %or, equivalently
```

(b) Let the input $u(t)$ in the state equation (P8.8.2) be dependent on the state as

$$u(t) = K\ \mathbf{x}(t) = [\, K_0 x_1(t) \quad K_1 x_2(t) \quad K_2 x_3(t)\,] \tag{P8.8.4}$$

Then, the state equation can be written as

$$\begin{bmatrix} x_1{}'(t) \\ x_2{}'(t) \\ x_3{}'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ K_0 - a_0 & K_1 - a_1 & K_2 - a_2 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} \tag{P8.8.5}$$

If the parameters of the original system matrix are $a_0 = 1, a_1 = -2$, and $a_2 = 3$, what are the values of the gain matrix $K = [K_0\ K_1\ K_2]$ you will fix so that the virtual system matrix in the state equation (P8.8.5) has the eigenvalues of $\lambda = -1, -2$, and -3? Note that the characteristic equation of the system whose behavior is described by the state equation (P8.8.5) is

$$s^3 + (a_2 - K_2)s^2 + (a_1 - K_1)s + a_0 - K_0 = 0 \tag{P8.8.6}$$

and the equation having the roots of $\lambda = -1, -2$, and -3 is

$$(s + 1)(s + 2)(s + 3) = s^3 + 6s^2 + 11s + 6 = 0 \tag{P8.8.7}$$

8.9 A Homogeneous Differential Equation—An Eigenvalue Equation

Consider the undamped mass-spring system depicted in Fig. 8.3, where the masses and the spring constants are $m_1 = 1, m_2 = 1$[kg] and $k_1 = 5, k_2 = 10$ [N/m], respectively. Complete the following program "`nm8p09.m`" whose objective is to solve the second-order differential equation (8.6.1) with the initial conditions $[x_1(0), x_2(0), x_1'(0), x_2'(0)] = [1, -0.5, 0, 0]$ for the time interval [0,10] in two ways—that is, by using the ODE-solver "`ode45()`" (Section 6.5.1) and by using the eigenvalue method (Section 8.6) and plot the two solutions. Run the completed program to obtain the solution graphs for $x_1(t)$ and $x_2(t)$.

(cf) Note that the second-order vector differential equation (8.6.1) can be written as the following state equation:

$$\begin{bmatrix} \mathbf{x}'(t) \\ \mathbf{x}''(t) \end{bmatrix} = \begin{bmatrix} O & I \\ -A & O \end{bmatrix} \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{x}'(t) \end{bmatrix} \tag{P8.9.1}$$

```
%nm8p09.m solve a set of differential eqs. (a state equation)
clear, clf
global A
df = 'df861';
k1 = 5; k2 = 10; m1 = 1; m2 = 1; % the spring constants and the masses
A = [(k1 + k2)/m1 -k2/m1; -k2/m2 k2/m2]; NA = size(A,2);
t0 = 0; tf =??; x0 =[? ???? ? ?]; % initial/final time, initial values
[t4,x4] = ode45(df,[t0 tf],x0);
[V,LAMBDA] = eig(A); % modal matrix composed of eigenvectors
w0 = x0(1:NA)*V; w10 = x0(NA+1:end)*V; % Eq.(8.6.8)
omega = ??????????????????;
for n = 1:NA % Eq.(8.6-7)
  omegan=omega(n);
  w(:,n) = [cos(omega n;*t4) sin(omega n*t4)]*[w0(n);w10(n)/omega n];
end
xE = w*V.'; % Eq.(8.6.3)
for n = 1:NA
  subplot(311 + n), plot(t4,x4(:,n),'b', t4,xE(:,n),'r')
end
```

```
function dx = df861(t,x)
global A
NA = size(A,2);
if length(x) ~= 2*NA, error('Some dimension problem'); end
dx = [zeros(NA) eye(NA); -A zeros(NA)]*x(:);
if size(x,2) > 1, dx = dx.'; end
```

9

PARTIAL DIFFERENTIAL EQUATIONS

What is a partial differential equation (PDE)? It is a class of differential equations involving more than one independent variable. In this chapter, we consider a general second-order PDE in two independent variables x and y, which is written as

$$A(x, y)\frac{\partial^2 u}{\partial x^2} + B(x, y)\frac{\partial^2 u}{\partial x \partial y} + C(x, y)\frac{\partial^2 u}{\partial y^2} = f\left(x, y, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right) \tag{9.0.1}$$
$$\text{for } x_0 \le x \le x_f,\ y_0 \le y \le y_f$$

with the boundary conditions given by

$$\begin{aligned} &u(x, y_0) = b_{y0}(x), \quad u(x, y_f) = b_{yf}(x), \\ &u(x_0, y) = b_{x0}(y), \quad \text{and} \quad u(x_f, y) = b_{xf}(y) \end{aligned} \tag{9.0.2}$$

These PDEs are classified into three groups:

$$\begin{aligned} &\text{Elliptic PDE:} \quad \text{if } B^2 - 4AC < 0 \\ &\text{Parabolic PDE:} \quad \text{if } B^2 - 4AC = 0 \\ &\text{Hyperbolic PDE:} \quad \text{if } B^2 - 4AC > 0 \end{aligned}$$

These three types of PDE are associated with equilibrium states, diffusion states, and oscillating systems, respectively. We will study some numerical methods for solving these PDEs, since their analytical solutions are usually difficult to find.

9.1 ELLIPTIC PDE

As an example, we will deal with a special type of elliptic equation called Helmholtz's equation, which is written as

$$\nabla^2 u(x,y) + g(x,y)u(x,y) = \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} + g(x,y)u(x,y) = f(x,y) \tag{9.1.1}$$

over a domain $D = \{(x,y) | x_0 \le x \le x_f, y_0 \le y \le y_f\}$ with some boundary conditions of

$$\begin{aligned} u(x_0, y) &= b_{x0}(y), \quad u(x_f, y) = b_{xf}(y), \\ u(x, y_0) &= b_{y0}(x), \quad \text{and} \quad u(x, y_f) = b_{yf}(x) \end{aligned} \tag{9.1.2}$$

(cf) Equation (9.1.1) is called Poisson's equation if $g(x,y) = 0$ and it is called Laplace's equation if $g(x,y) = 0$ and $f(x,y) = 0$.

To apply the difference method, we divide the domain into M_x sections, each of length $\Delta x = (x_f - x_0)/M_x$ along the x-axis and into M_y sections, each of length $\Delta y = (y_f - y_0)/M_y$ along the y-axis, respectively, and then replace the second derivatives by the three-point central difference approximation (5.3.1)

$$\left.\frac{\partial^2 u(x,y)}{\partial x^2}\right|_{x_j, y_i} \cong \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta x^2} \quad \text{with } x_j = x_0 + j\Delta x,\ y_i = y_0 + i\Delta y \tag{9.1.3a}$$

$$\left.\frac{\partial^2 u(x,y)}{\partial y^2}\right|_{x_j, y_i} \cong \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta y^2} \quad \text{with } u_{i,j} = u(x_j, y_i) \tag{9.1.3b}$$

so that, for every interior point (x_j, y_i) with $1 \le i \le M_y - 1$ and $1 \le j \le M_x - 1$, we obtain the finite difference equation

$$\frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta x^2} + \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta y^2} + g_{i,j}u_{i,j} = f_{i,j} \tag{9.1.4}$$

where

$$u_{i,j} = u(x_j, y_i), \quad f_{i,j} = f(x_j, y_i), \quad \text{and} \quad g_{i,j} = g(x_j, y_i)$$

These equations can somehow be arranged into a system of simultaneous equations with respect to the $(M_y - 1)(M_x - 1)$ variables $\{u_{1,1}, u_{1,2}, \ldots, u_{1,M_x-1}, u_{2,1}, \ldots, u_{2,M_x-1}, \ldots, u_{M_y-1,1}, u_{M_y-1,2}, \ldots, u_{M_y-1,M_x-1}\}$, but it seems to be messy to work with and we may be really in trouble as M_x and M_y become large. A simpler way is to use the iterative methods introduced in Section 2.5. To do so, we first need to shape the equations and the boundary conditions into the following form:

$$u_{i,j} = r_y(u_{i,j+1} + u_{i,j-1}) + r_x(u_{i+1,j} + u_{i-1,j}) + r_{xy}(g_{i,j}u_{i,j} - f_{i,j}) \tag{9.1.5a}$$

$$u_{i,0} = b_{x0}(y_i), \quad u_{i,M_x} = b_{xf}(y_i), \quad u_{0,j} = b_{y0}(x_j), \quad u_{M_y,j} = b_{yf}(x_j) \quad (9.1.5b)$$

where

$$\frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)} = r_y, \quad \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)} = r_x, \quad \frac{\Delta x^2 \Delta y^2}{2(\Delta x^2 + \Delta y^2)} = r_{xy} \quad (9.1.6)$$

How do we initialize this algorithm? If we have no priori knowledge about the solution, it is reasonable to take the average value of the boundary values as the initial values of $u_{i,j}$.

The objective of the MATLAB routine "`poisson.m`" is to solve the above equation.

```
function [u,x,y] = poisson(f,g,bx0,bxf,by0,byf,D,Mx,My,tol,MaxIter)
%solve u_xx + u_yy + g(x,y)u = f(x,y)
%  over the region D = [x0,xf,y0,yf] = {(x,y) |x0 <= x <= xf, y0 <= y <= yf}
% with the boundary Conditions:
%  u(x0,y) = bx0(y), u(xf,y)  = bxf(y)
%  u(x,y0) = by0(x), u(x,yf)  = byf(x)
% Mx = # of subintervals along x axis
% My = # of subintervals along y axis
% tol     : error tolerance
% MaxIter: the maximum # of iterations
x0 = D(1); xf = D(2); y0 = D(3); yf = D(4);
dx = (xf - x0)/Mx;  x = x0 + [0:Mx]*dx;
dy = (yf - y0)/My;  y = y0 + [0:My]'*dy;
Mx1 = Mx + 1; My1 = My + 1;
%Boundary conditions
for m = 1:My1, u(m,[1 Mx1])=[bx0(y(m)) bxf(y(m))]; end %left/right side
for n = 1:Mx1, u([1 My1],n) = [by0(x(n)); byf(x(n))]; end %bottom/top
%initialize as the average of boundary values
sum_of_bv = sum(sum([u(2:My,[1 Mx1]) u([1 My1],2:Mx)']));
u(2:My,2:Mx) = sum_of_bv/(2*(Mx + My - 2));
for i = 1:My
  for j = 1:Mx
     F(i,j) = f(x(j),y(i)); G(i,j) = g(x(j),y(i));
  end
end
dx2 = dx*dx; dy2 = dy*dy;  dxy2 = 2*(dx2 + dy2);
rx = dx2/dxy2; ry = dy2/dxy2;  rxy = rx*dy2;
for itr = 1:MaxIter
 for j = 2:Mx
  for i = 2:My
    u(i,j) = ry*(u(i,j + 1)+u(i,j - 1)) + rx*(u(i + 1,j)+u(i - 1,j))...
               + rxy*(G(i,j)*u(i,j)- F(i,j)); %Eq.(9.1.5a)
  end
 end
 if itr > 1 & max(max(abs(u - u0))) < tol, break; end
 u0 = u;
end
```

```
%solve_poisson in Example 9.1
f = inline('0','x','y');  g = inline('0','x','y');
x0 = 0; xf = 4; Mx = 20;  y0 = 0; yf = 4; My = 20;
bx0 = inline('exp(y) - cos(y)','y');               %(E9.1.2a)
bxf = inline('exp(y)*cos(4) - exp(4)*cos(y)','y'); %(E9.1.2b)
by0 = inline('cos(x) - exp(x)','x');               %(E9.1.3a)
byf = inline('exp(4)*cos(x) - exp(x)*cos(4)','x'); %(E9.1.3b)
D = [x0 xf y0 yf];  MaxIter = 500;  tol = 1e-4;
[U,x,y] = poisson(f,g,bx0,bxf,by0,byf,D,Mx,My,tol,MaxIter);
clf, mesh(x,y,U), axis([0 4 0 4 -100 100])
```

Example 9.1. Laplace's Equation—Steady-State Temperature Distribution.
Consider Laplace's equation

$$\nabla^2 u(x, y) = \frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = 0 \qquad \text{for } 0 \le x \le 4, 0 \le y \le 4 \tag{E9.1.1}$$

with the boundary conditions

$$u(0, y) = e^y - \cos y, \qquad u(4, y) = e^y \cos 4 - e^4 \cos y \tag{E9.1.2}$$

$$u(x, 0) = \cos x - e^x, \qquad u(x, 4) = e^4 \cos x - e^x \cos 4 \tag{E9.1.3}$$

What we will get from solving this equation is $u(x, y)$, which supposedly describes the temperature distribution over a square plate having each side 4 units long (Fig. 9.1). We made the MATLAB program "`solve_poisson.m`" in order to use the routine "`poisson()`" to solve Laplace's equation given above and run this program to obtain the result shown in Fig. 9.2.

Now, let us consider the so-called Neumann boundary conditions described as

$$\left.\frac{\partial u(x, y)}{\partial x}\right|_{x=x_0} = b'_{x_0}(y) \qquad \text{for } x = x_0 \text{ (the left-side boundary)} \tag{9.1.7}$$

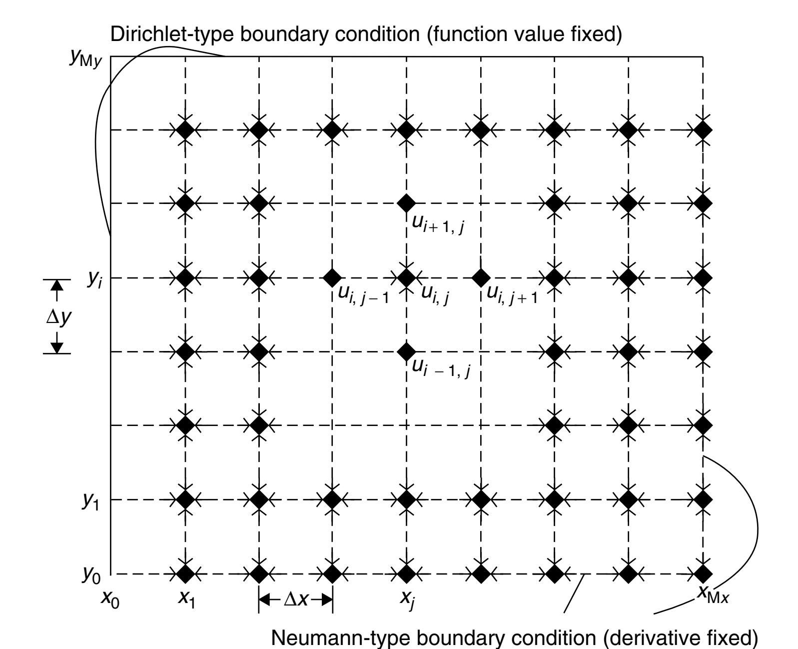


Figure 9.1 The grid for elliptic equations with Dirichlet/Neumann-type boundary condition.

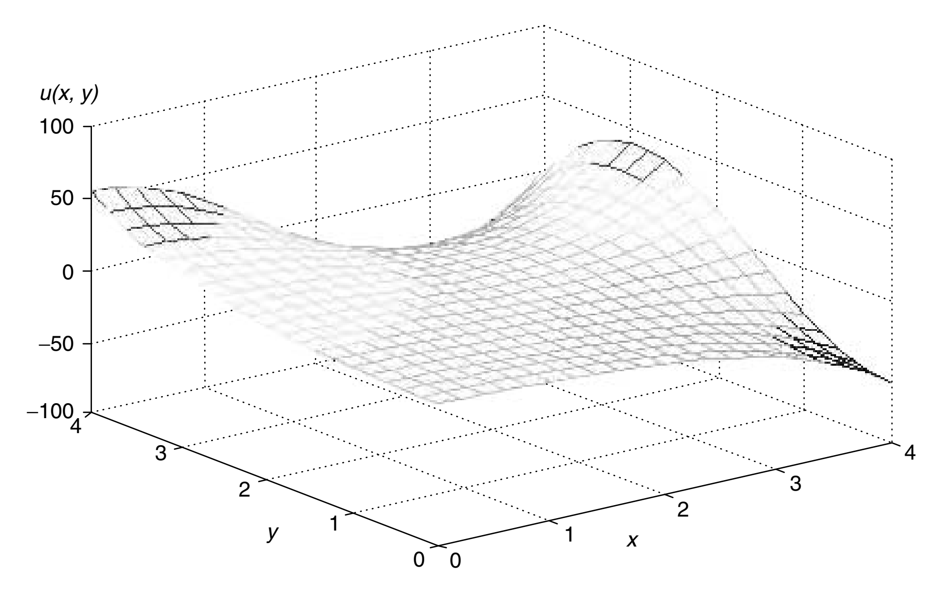


Figure 9.2 Temperature distribution over a plate — Example 9.1.

Replacing the first derivative on the left-side boundary ($x = x_0$) by its three-point central difference approximation (5.1.8)

$$\frac{u_{i,1} - u_{i,-1}}{2\Delta x} \approx b'_{x_0}(y_i), \quad u_{i,-1} \approx u_{i,1} - 2b'_{x_0}(y_i)\Delta x \quad \text{for } i = 1, 2, \ldots, M_y - 1 \tag{9.1.8}$$

and then substituting this constraint into Eq. (9.1.5a) at the boundary points, we have

$$\begin{aligned} u_{i,0} &= r_y(u_{i,1} + u_{i,-1}) + r_x(u_{i+1,0} + u_{i-1,0}) + r_{xy}(g_{i,0}u_{i,0} - f_{i,0}) \\ &= r_y(u_{i,1} + u_{i,1} - 2b'_{x_0}(y_i)\Delta x) + r_x(u_{i+1,0} + u_{i-1,0}) + r_{xy}(g_{i,0}u_{i,0} - f_{i,0}) \\ &= 2r_y u_{i,1} + r_x(u_{i+1,0} + u_{i-1,0}) + r_{xy}(g_{i,0}u_{i,0} - f_{i,0} - 2b'_{x_0}(y_i)/\Delta x) \\ &\qquad \text{for } i = 1, 2, \ldots, M_y - 1 \end{aligned} \tag{9.1.9}$$

If the boundary condition on the lower side boundary ($y = y_0$) is also of Neumann type, then we need to write similar equations for $j = 1, 2, \ldots, M_x - 1$

$$u_{0,j} = r_y(u_{0,j+1} + u_{0,j-1}) + 2r_x u_{1,j} + r_{xy}(g_{0,j}u_{0,j} - f_{0,j} - 2b'_{y_0}(x_j)/\Delta y) \tag{9.1.10}$$

and additionally for the left-lower corner point (x_0, y_0),

$$u_{0,0} = 2(r_y u_{0,1} + r_x u_{1,0}) + r_{xy}(g_{0,0}u_{0,0} - f_{0,0} - 2(b'_{x_0}(y_0)/\Delta x + 2b'_{y_0}(x_0)/\Delta y)) \tag{9.1.11}$$

9.2 PARABOLIC PDE

An example of a parabolic PDE is a one-dimensional heat equation describing the temperature distribution $u(x, t)$ (x is position, t is time) as

$$A\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \qquad \text{for } 0 \le x \le x_f, \quad 0 \le t \le T \tag{9.2.1}$$

In order for this equation to be solvable, the boundary conditions $u(0, t) = b_0(t)$ & $u(x_f, t) = b_{xf}(t)$ as well as the initial condition $u(x, 0) = i_0(x)$ should be provided.

9.2.1 The Explicit Forward Euler Method

To apply the finite difference method, we divide the spatial domain $[0, x_f]$ into M sections, each of length $\Delta x = x_f/M$, and divide the time domain $[0, T]$ into N segments, each of duration $\Delta t = T/N$, and then replace the second partial derivative on the left-hand side and the first partial derivative on the right-hand side of the above equation (9.2.1) by the central difference approximation (5.3.1) and the forward difference approximation (5.1.4), respectively, so that we have

$$A\frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{\Delta x^2} = \frac{u_i^{k+1} - u_i^k}{\Delta t} \tag{9.2.2}$$

This can be cast into the following algorithm, called the explicit forward Euler method, which is to be solved iteratively:

$$u_i^{k+1} = r(u_{i+1}^k + u_{i-1}^k) + (1 - 2r)u_i^k \qquad \text{with } r = A\frac{\Delta t}{\Delta x^2} \tag{9.2.3}$$
$$\text{for } i = 1, 2, \ldots, M - 1$$

To find the stability condition of this algorithm, we substitute a trial solution

$$u_i^k = \lambda^k e^{ji\pi/P} (P \text{ is any nonzero integer}) \tag{9.2.4}$$

into Eq. (9.2.3) to get

$$\lambda = r(e^{j\pi/P} + e^{-j\pi/P}) + (1 - 2r) = 1 - 2r(1 - \cos(\pi/P)) \tag{9.2.5}$$

Since we must have $|\lambda| \le 1$ for nondivergence, the stability condition turns out to be

$$r = A\frac{\Delta t}{\Delta x^2} \le \frac{1}{2} \tag{9.2.6}$$

```
function [u,x,t] = heat_exp(a,xf,T,it0,bx0,bxf,M,N)
%solve a u_xx = u_t  for 0 <= x <= xf, 0 <= t <= T
% Initial Condition: u(x,0) = it0(x)
% Boundary Condition: u(0,t) = bx0(t), u(xf,t) = bxf(t)
% M = # of subintervals along x axis
% N = # of subintervals along t axis
dx = xf/M;  x = [0:M]'*dx;
dt = T/N;  t = [0:N]*dt;
for i = 1:M + 1, u(i,1) = it0(x(i)); end
for n = 1:N + 1, u([1 M + 1],n) = [bx0(t(n)); bxf(t(n))]; end
r = a*dt/dx/dx,  r1 = 1 - 2*r;
for k = 1:N
   for i = 2:M
      u(i,k+1) = r*(u(i + 1,k) + u(i-1,k)) + r1*u(i,k); %Eq.(9.2.3)
   end
end
```

This implies that as we decrease the spatial interval Δx for better accuracy, we must also decrease the time step Δt at the cost of more computations in order not to lose the stability.

The MATLAB routine "heat_exp()" has been composed to implement this algorithm.

9.2.2 The Implicit Backward Euler Method

In this section, we consider another algorithm called the implicit backward Euler method, which comes out from substituting the backward difference approximation (5.1.6) for the first partial derivative on the right-hand side of Eq. (9.2.1) as

$$A\frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{\Delta x^2} = \frac{u_i^k - u_i^{k-1}}{\Delta t} \tag{9.2.7}$$

$$-ru_{i-1}^k + (1+2r)u_i^k - ru_{i+1}^k = u_i^{k-1} \qquad \text{with } r = A\frac{\Delta t}{\Delta x^2} \tag{9.2.8}$$

$$\text{for } i = 1, 2, \ldots, M-1$$

If the values of u_0^k and u_M^k at both end points are given from the Dirichlet type of boundary condition, then the above equation will be cast into a system of simultaneous equations:

$$\begin{bmatrix} 1+2r & -r & 0 & \cdot & 0 & 0 \\ -r & 1+2r & -r & \cdot & 0 & 0 \\ 0 & -r & 1+2r & \cdot & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & 1+2r & -r \\ 0 & 0 & 0 & \cdot & -r & 1+2r \end{bmatrix} \begin{bmatrix} u_1^k \\ u_2^k \\ u_3^k \\ \cdot \\ u_{M-2}^k \\ u_{M-1}^k \end{bmatrix} = \begin{bmatrix} u_1^{k-1} + ru_0^k \\ u_2^{k-1} \\ u_3^{k-1} \\ \cdot \\ u_{M-2}^{k-1} \\ u_{M-1}^{k-1} + ru_M^k \end{bmatrix} \tag{9.2.9}$$

How about the case where the values of $\partial u/\partial x|_{x=0} = b_0'(t)$ at one end are given? In that case, we approximate this Neumann type of boundary condition by

$$\frac{u_1^k - u_{-1}^k}{2\Delta x} = b_0'(k) \tag{9.2.10}$$

and mix it up with one more equation associated with the unknown variable u_0^k

$$-ru_{-1}^k + (1+2r)u_0^k - ru_1^k = u_0^{k-1} \tag{9.2.11}$$

to get

$$(1+2r)u_0^k - 2ru_1^k = u_0^{k-1} - 2rb_0'(k)\Delta x \tag{9.2.12}$$

We augment Eq. (9.2.9) with this to write

$$\begin{bmatrix} 1+2r & -2r & 0 & 0 & \cdot & 0 & 0 \\ -r & 1+2r & -r & 0 & \cdot & 0 & 0 \\ 0 & -r & 1+2r & -r & \cdot & 0 & 0 \\ 0 & 0 & -r & 1+2r & \cdot & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot & 1+2r & -r \\ 0 & 0 & 0 & \cdot & \cdot & -r & 1+2r \end{bmatrix} \begin{bmatrix} u_0^k \\ u_1^k \\ u_2^k \\ u_3^k \\ \cdot \\ u_{M-2}^k \\ u_{M-1}^k \end{bmatrix} = \begin{bmatrix} u_0^{k-1} - 2rb_0'(k)\Delta x \\ u_1^{k-1} \\ u_2^{k-1} \\ u_3^{k-1} \\ \cdot \\ u_{M-2}^{k-1} \\ u_{M-1}^{k-1} + ru_M^k \end{bmatrix} \tag{9.2.13}$$

Equations such as Eq. (9.2.9) or (9.2.13) are really nice in the sense that they can be solved very efficiently by exploiting their tridiagonal structures and are guaranteed to be stable owing to their diagonal dominancy. The unconditional stability of Eq. (9.2.9) can be shown by substituting Eq. (9.2.4) into Eq. (9.2.8):

$$-re^{-j\pi/P} + (1+2r) - re^{j\pi/P} = 1/\lambda, \ \ \lambda = \frac{1}{1+2r(1-\cos(\pi/P))}, \ \ |\lambda| \le 1 \tag{9.2.14}$$

The following routine "heat_imp()" implements this algorithm to solve the PDE (9.2.1) with the ordinary (Dirichlet type of) boundary condition via Eq. (9.2.9).

```
function [u,x,t] = heat_imp(a,xf,T,it0,bx0,bxf,M,N)
%solve a u_xx = u_t  for 0 <= x <= xf, 0 <= t <= T
% Initial Condition: u(x,0) = it0(x)
% Boundary Condition: u(0,t) = bx0(t), u(xf,t) = bxf(t)
% M = # of subintervals along x axis
% N = # of subintervals along t axis
dx = xf/M;  x = [0:M]'*dx;
dt = T/N;  t = [0:N]*dt;
for i = 1:M + 1, u(i,1) = it0(x(i)); end
for n = 1:N + 1, u([1 M + 1],n) = [bx0(t(n)); bxf(t(n))]; end
r = a*dt/dx/dx;      r2 = 1 + 2*r;
for i = 1:M - 1
   A(i,i) = r2; %Eq.(9.2.9)
   if i > 1, A(i - 1,i) = -r; A(i,i - 1) = -r; end
end
for k = 2:N + 1
   b = [r*u(1,k); zeros(M - 3,1); r*u(M + 1,k)] + u(2:M,k - 1); %Eq.(9.2.9)
   u(2:M,k) = trid(A,b);
end
```

9.2.3 The Crank–Nicholson Method

Here, let us go back to see Eq. (9.2.7) and try to improve the implicit backward Euler method. The difference approximation on the left-hand side is taken at time point k, while the difference approximation on the right-hand side is taken at the midpoint between time k and $k-1$, if we regard it as the central difference approximation with time step $\Delta t/2$. Doesn't this seem to be inconsistent? How about taking the difference approximation of both sides at the same time point—say, the midpoint between $k+1$ and k—for balance? In order to do so, we take the average of the central difference approximations of the left-hand side at the two points $k+1$ and k, yielding

$$\frac{A}{2}\left(\frac{u_{i+1}^{k+1}-2u_i^{k+1}+u_{i-1}^{k+1}}{\Delta x^2}+\frac{u_{i+1}^{k}-2u_i^{k}+u_{i-1}^{k}}{\Delta x^2}\right)=\frac{u_i^{k+1}-u_i^k}{\Delta t} \tag{9.2.15}$$

which leads to the so-called Crank–Nicholson method:

$$-ru_{i+1}^{k+1}+2(1+r)u_i^{k+1}-ru_{i-1}^{k+1}=ru_{i+1}^{k}+2(1-r)u_i^{k}+ru_{i-1}^{k} \tag{9.2.16}$$

$$\text{with } r=A\frac{\Delta t}{\Delta x^2}$$

With the Dirichlet/Neumann type of boundary condition on x_0/x_M, respectively, this can be cast into the following tridiagonal system of equations.

$$\begin{bmatrix} 2(1+r) & -r & 0 & \cdot & 0 & 0 \\ -r & 2(1+r) & -r & \cdot & 0 & 0 \\ 0 & -r & 2(1+r) & \cdot & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & 2(1+r) & -r \\ 0 & 0 & 0 & \cdot & -2r & 2(1+r) \end{bmatrix}\begin{bmatrix} u_1^{k+1} \\ u_2^{k+1} \\ u_3^{k+1} \\ \cdot \\ u_{M-1}^{k+1} \\ u_M^{k+1} \end{bmatrix}$$

$$= \begin{bmatrix} 2(1-r) & r & 0 & \cdot & 0 & 0 \\ r & 2(1-r) & r & \cdot & 0 & 0 \\ 0 & r & 2(1-r) & \cdot & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & 2(1-r) & r \\ 0 & 0 & 0 & \cdot & 2r & 2(1-r) \end{bmatrix}\begin{bmatrix} u_1^{k} \\ u_2^{k} \\ u_3^{k} \\ \cdot \\ u_{M-1}^{k} \\ u_M^{k} \end{bmatrix} + \begin{bmatrix} r(u_0^{k+1}+u_0^k) \\ 0 \\ 0 \\ \cdot \\ 0 \\ 2r(b'_M(k+1)+b'_M(k)) \end{bmatrix} \tag{9.2.17}$$

This system of equations can also be solved very efficiently, and its unconditional stability can be shown by substituting Eq. (9.2.4) into Eq. (9.2.16):

$$2\lambda(1+r(1-\cos(\pi/P)))=2(1-r(1-\cos(\pi/P))),$$

$$\lambda=\frac{1-r(1-\cos(\pi/P))}{1+r(1-\cos(\pi/P))},\qquad |\lambda|\le 1 \tag{9.2.18}$$

This algorithm is cast into the following MATLAB routine "`heat_CN()`".

```
function [u,x,t] = heat_CN(a,xf,T,it0,bx0,bxf,M,N)
%solve a u_xx = u_t  for 0 <= x <= xf, 0 <= t <= T
% Initial Condition: u(x,0) = it0(x)
% Boundary Condition: u(0,t) = bx0(t), u(xf,t) = bxf(t)
% M = # of subintervals along x axis
% N = # of subintervals along t axis
dx = xf/M;  x = [0:M]'*dx;
dt = T/N;  t = [0:N]*dt;
for i = 1:M + 1, u(i,1) = it0(x(i)); end
for n = 1:N + 1, u([1 M + 1],n) = [bx0(t(n)); bxf(t(n))]; end
r = a*dt/dx/dx;
r1 = 2*(1 - r); r2 = 2*(1 + r);
for i = 1:M - 1
   A(i,i) = r1; %Eq.(9.2.17)
   if i > 1, A(i - 1,i) = -r; A(i,i - 1) = -r; end
end
for k = 2:N + 1
    b = [r*u(1,k); zeros(M - 3,1); r*u(M + 1,k)] ...
          + r*(u(1:M - 1,k - 1) + u(3:M + 1,k - 1)) + r2*u(2:M,k - 1);
    u(2:M,k) = trid(A,b); %Eq.(9.2.17)
end
```

Example 9.2. One-Dimensional Parabolic PDE: Heat Flow Equation.
Consider the parabolic PDE

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \qquad \text{for } 0 \le x \le 1, 0 \le t \le 0.1 \tag{E9.2.1}$$

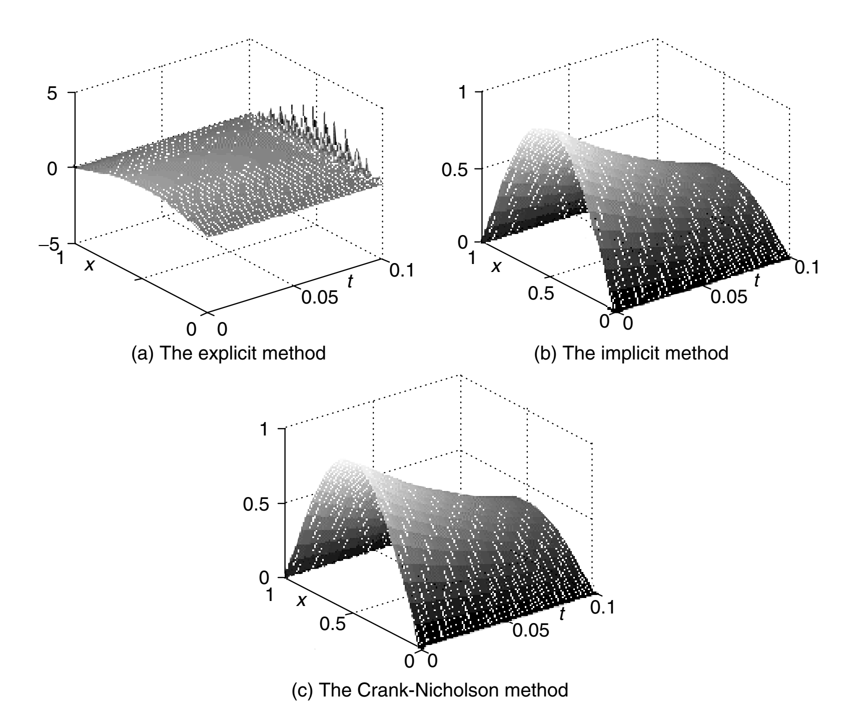
with the initial condition and the boundary conditions

$$u(x,0) = \sin \pi x, \qquad u(0,t) = 0, \qquad u(1,t) = 0 \tag{E9.2.2}$$

We made the MATLAB program "`solve_heat.m`" in order to use the routines "`heat_exp()`", "`heat_imp()`", and "`heat_CN()`" in solving this equation and ran this program to obtain the results shown in Fig. 9.3. Note that with the spatial interval $\Delta x = x_f/M = 1/20$ and the time step $\Delta t = T/N = 0.1/100 = 0.001$, we have

$$r = A\frac{\Delta t}{\Delta x^2} = 1\frac{0.001}{(1/20)^2} = 0.4 \tag{E9.2.3}$$

which satisfies the stability condition ($r \le 1/2$) (9.2.6) and all of the three methods lead to reasonably fair results with a relative error of about 0.013. But, if we decrease the spatial interval to $\Delta x = 1/25$ for better resolution, we have $r = 0.625$, violating the stability condition and the explicit forward Euler method ("`heat_exp()`") blows up because of instability as shown in Fig. 9.3a, while the implicit backward Euler method ("`heat_imp()`") and the Crank–Nicholson method ("`heat_CN()`") work quite well as shown in Figs. 9.3b,c. Now, with the spatial interval $\Delta x = 1/25$ and the time step $\Delta t = 0.1/120$, the explicit method as well as the other ones works well with a relative error less than 0.001 in return



(a) The explicit method (b) The implicit method

(c) The Crank-Nicholson method

Figure 9.3 Results of various algorithms for a one-dimensional parabolic PDE: heat equation.

for somewhat (30%) more computations, despite that $r = 0.5208$ doesn't strictly satisfy the stability condition.

This implies that the condition ($r \leq 1/2$) for stability of the explicit forward Euler method is not a necessary one, but only a sufficient one. Besides, if it converges, its accuracy may be better than that of the implicit backward Euler method, but generally no better than that of the Crank–Nicholson method.

```
%solve_heat
a = 1; %the parameter of (E9.2.1)
it0 = inline('sin(pi*x)','x'); %initial condition
bx0 = inline('0');  bxf = inline('0'); %boundary condition
xf = 1; M = 25;  T = 0.1; N = 100; %r = 0.625
%analytical solution
uo = inline('sin(pi*x)*exp(-pi*pi*t)','x','t');
[u1,x,t] = heat_exp(a,xf,T,it0,bx0,bxf,M,N);
figure(1), clf, mesh(t,x,u1)
[u2,x,t] = heat_imp(a,xf,T,it0,bx0,bxf,M,N); %converge unconditionally
figure(2), clf, mesh(t,x,u2)
[u3,x,t] = heat_CN(a,xf,T,it0,bx0,bxf,M,N); %converge unconditionally
figure(3), clf, mesh(t,x,u3)
MN = M*N;
Uo = uo(x,t); aUo = abs(Uo)+eps; %values of true analytical solution
%How far from the analytical solution?
err1 = norm((u1-Uo)./aUo)/MN
err2 = norm((u2-Uo)./aUo)/MN
err3 = norm((u3-Uo)./aUo)/MN
```

9.2.4 Two-Dimensional Parabolic PDE

Another example of a parabolic PDE is a two-dimensional heat equation describing the temperature distribution $u(x, y, t)((x, y)$ is position, t is time) as

$$A\left(\frac{\partial^2 u(x, y, t)}{\partial x^2} + \frac{\partial^2 u(x, y, t)}{\partial y^2}\right) = \frac{\partial u(x, y, t)}{\partial t} \tag{9.2.19}$$

$$\text{for } x_0 \le x \le x_f,\ y_0 \le y \le y_f,\ 0 \le t \le T$$

In order for this equation to be solvable, we should be provided with the boundary conditions

$$u(x_0, y, t) = b_{x0}(y, t), \quad u(x_f, y, t) = b_{xf}(y, t),$$

$$u(x, y_0, t) = b_{y0}(x, t), \quad \text{and} \quad u(x, y_f, t) = b_{yf}(x, t)$$

as well as the initial condition $u(x, y, 0) = i_0(x, y)$.

We replace the first-order time derivative on the right-hand side by the three-point central difference at the midpoint $(t_{k+1} + t_k)/2$ just as with the Crank–Nicholson method. We also replace one of the second-order derivatives, u_{xx} and u_{yy}, by the three-point central difference approximation (5.3.1) at time t_k and the other at time t_{k+1}, yielding

$$A\left(\frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{\Delta x^2} + \frac{u_{i+1,j}^{k+1} - 2u_{i,j}^{k+1} + u_{i-1,j}^{k+1}}{\Delta y^2}\right) = \frac{u_{i,j}^{k+1} - u_{i,j}^k}{\Delta t} \tag{9.2.20}$$

which seems to be attractive, since it can be formulated into a tridiagonal system of equations with respect to $u_{i+1,j}^{k+1}$, $u_{i,j}^{k+1}$, and $u_{i-1,j}^{k+1}$. But, why do we treat u_{xx} and u_{yy} with discrimination—that is, evaluate one at time t_k and the other at time t_{k+1} in a fixed manner? In an alternate manner, we write the difference equation for the next time point t_{k+1} as

$$A\left(\frac{u_{i,j+1}^{k+1} - 2u_{i,j}^{k+1} + u_{i,j-1}^{k+1}}{\Delta x^2} + \frac{u_{i+1,j}^{k} - 2u_{i,j}^{k} + u_{i-1,j}^{k}}{\Delta y^2}\right) = \frac{u_{i,j}^{k+2} - u_{i,j}^{k+1}}{\Delta t} \tag{9.2.21}$$

This formulation, proposed by Peaceman and Rachford [P-1], is referred to as the alternating direction implicit (ADI) method and can be cast into the following algorithm:

$$-r_y(u_{i-1,j}^{k+1} + u_{i+1,j}^{k+1}) + (1 + 2r_y)u_{i,j}^{k+1} = r_x(u_{i,j-1}^k + u_{i,j+1}^k) + (1 - 2r_x)u_{i,j}^k$$

$$\text{for } 1 \le j \le M_x - 1 \tag{9.2.22a}$$

$$-r_x(u_{i,j-1}^{k+2} + u_{i,j+1}^{k+2}) + (1 + 2r_x)u_{i,j}^{k+2} = r_y(u_{i-1,j}^{k+1} + u_{i+1,j}^{k+1}) + (1 - 2r_y)u_{i,j}^{k+1}$$

$$\text{for } 1 \le i \le M_y - 1 \tag{9.2.22b}$$

with

$$r_x = A\Delta t/\Delta x^2, \quad r_y = A\Delta t/\Delta y^2,$$
$$\Delta x = (x_f - x_0)/M_x, \quad \Delta y = (y_f - y_0)/M_y, \quad \Delta t = T/N$$

The objective of the following MATLAB routine "`heat2_ADI()`" is to implement this algorithm for solving a two-dimensional heat equation (9.2.19).

```
function [u,x,y,t] = heat2_ADI(a,D,T,ixy0,bxyt,Mx,My,N)
%solve u_t = c(u_xx + u_yy) for D(1) <= x <= D(2), D(3) <= y <= D(4), 0 <= t <= T
% Initial Condition: u(x,y,0) = ixy0(x,y)
% Boundary Condition: u(x,y,t) = bxyt(x,y,t) for (x,y)cB
% Mx/My = # of subintervals along x/y axis
% N = # of subintervals along t axis
dx = (D(2) - D(1))/Mx; x = D(1)+[0:Mx]*dx;
dy = (D(4) - D(3))/My; y = D(3)+[0:My]'*dy;
dt = T/N;  t = [0:N]*dt;
%Initialization
for j = 1:Mx + 1
  for i = 1:My + 1
    u(i,j) = ixy0(x(j),y(i));
  end
end
rx = a*dt/(dx*dx); rx1 = 1 + 2*rx; rx2 = 1 - 2*rx;
ry = a*dt/(dy*dy); ry1 = 1 + 2*ry; ry2 = 1 - 2*ry;
for j = 1:Mx - 1 %Eq.(9.2.22a)
  Ay(j,j) = ry1;
  if j > 1, Ay(j - 1,j) = -ry; Ay(j,j-1) = -ry; end
end
for i = 1:My - 1 %Eq.(9.2.22b)
  Ax(i,i) = rx1;
  if i > 1, Ax(i - 1,i) = -rx; Ax(i,i - 1) = -rx; end
end
for k = 1:N
  u_1 = u;   t = k*dt;
  for i = 1:My + 1 %Boundary condition
    u(i,1) = feval(bxyt,x(1),y(i),t);
    u(i,Mx+1) = feval(bxyt,x(Mx+1),y(i),t);
  end
  for j = 1:Mx + 1
    u(1,j) = feval(bxyt,x(j),y(1),t);
    u(My+1,j) = feval(bxyt,x(j),y(My + 1),t);
  end
  if mod(k,2) == 0
    for i = 2:My
      jj = 2:Mx;
      bx = [ry*u(i,1) zeros(1,Mx - 3) ry*u(i,My + 1)] ...
                      +rx*(u_1(i-1,jj)+ u_1(i + 1,jj)) + rx2*u_1(i,jj);
      u(i,jj) = trid(Ay,bx')'; %Eq.(9.2.22a)
    end
   else
    for j = 2:Mx
      ii = 2:My;
      by = [rx*u(1,j); zeros(My-3,1); rx*u(Mx + 1,j)] ...
                     + ry*(u_1(ii,j-1) + u_1(ii,j + 1)) + ry2*u_1(ii,j);
      u(ii,j) = trid(Ax,by); %Eq.(9.2.22b)
    end
  end
end
```

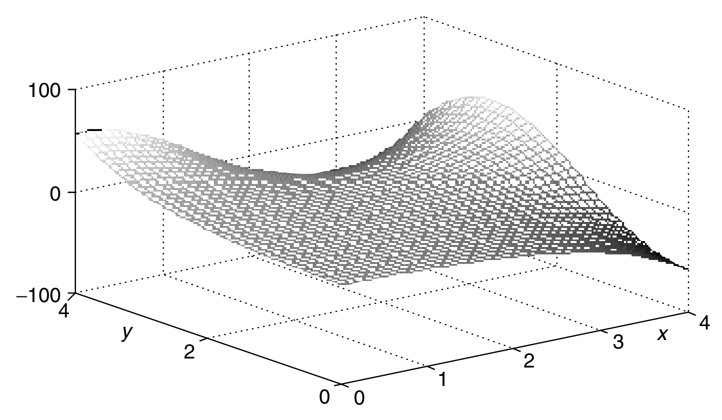


Figure 9.4 A solution for a two dimensional parabolic PDE obtained using "heat2_ADI()" (Example 9.3).

Example 9.3. A Parabolic PDE: Two-Dimensional Temperature Diffusion.
Consider a two-dimensional parabolic PDE

$$10^{-4}\left(\frac{\partial^2 u(x,y,t)}{\partial x^2}+\frac{\partial^2 u(x,y,t)}{\partial y^2}\right)=\frac{\partial u(x,y,t)}{\partial t} \tag{E9.3.1}$$

$$\text{for}\quad 0\le x\le 4,\quad 0\le y\le 4,\quad 0\le t\le 5000$$

with the initial conditions and boundary conditions

$$u(x,y,0)=0 \qquad \text{for } t=0 \tag{E9.3.2a}$$

$$u(x,y,t)=e^y\cos x-e^x\cos y \text{ for } x=0, x=4, y=0, y=4 \tag{E9.3.2b}$$

We made the following MATLAB program "solve_heat2.m" in order to use the routine "heat2_ADI()" to solve this equation and ran this program to get the result shown in Fig. 9.4 at the final time.

```
%solve_heat2
clear, clf
a = 1e-4;
it0 = inline('0','x','y'); %(E9.3.2a)
bxyt = inline('exp(y)*cos(x)-exp(x)*cos(y)','x','y','t'); %(E9.3.2b)
D = [0 4 0 4]; T = 5000; Mx = 40; My = 40; N = 50;
[u,x,y,t] = heat2_ADI(a,D,T,it0,bxyt,Mx,My,N);
mesh(x,y,u)
```

9.3 HYPERBOLIC PDE

An example of a hyperbolic PDE is a one-dimensional wave equation for the amplitude function $u(x,t)$(x is position, t is time) as

$$A\frac{\partial^2 u(x,t)}{\partial x^2}=\frac{\partial^2 u(x,t)}{\partial t^2} \quad \text{for } 0\le x\le x_f,\ 0\le t\le T \tag{9.3.1}$$

In order for this equation to be solvable, the boundary conditions $u(0, t) = b_0(t)$ and $u(x_f, t) = b_{xf}(t)$ as well as the initial conditions $u(x, 0) = i_0(x)$ and $\partial u/\partial t|_{t=0}(x, 0) = i'_0(x)$ should be provided.

9.3.1 The Explicit Central Difference Method

In the same way as with the parabolic PDEs, we replace the second derivatives on both sides of Eq. (9.3.1) by their three-point central difference approximation (5.3.1) as

$$A\frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{\Delta x^2} = \frac{u_i^{k+1} - 2u_i^k + u_i^{k-1}}{\Delta t^2} \quad \text{with } \Delta x = \frac{x_f}{M}, \Delta t = \frac{T}{N} \tag{9.3.2}$$

which leads to the explicit central difference method:

$$u_i^{k+1} = r(u_{i+1}^k + u_{i-1}^k) + 2(1 - r)u_i^k - u_i^{k-1} \qquad \text{with } r = A\frac{\Delta t^2}{\Delta x^2} \tag{9.3.3}$$

Since $u_i^{-1} = u(x_i, -\Delta t)$ is not given, we cannot get u_i^1 directly from this formula (9.3.3) with $k = 0$:

$$u_i^1 = r(u_{i+1}^0 + u_{i-1}^0) + 2(1 - r)u_i^0 - u_i^{-1} \tag{9.3.4}$$

Therefore, we approximate the initial condition on the derivative by the central difference as

$$\frac{u_i^1 - u_i^{-1}}{2\Delta t} = i'_0(x_i) \tag{9.3.5}$$

and make use of this to remove u_i^{-1} from Eq. (9.3.3):

$$u_i^1 = r(u_{i+1}^0 + u_{i-1}^0) + 2(1 - r)u_i^0 - (u_i^1 - 2i'_0(x_i)\Delta t)$$

$$u_i^1 = \frac{1}{2}r(u_{i+1}^0 + u_{i-1}^0) + (1 - r)u_i^0 + i'_0(x_i)\Delta t \tag{9.3.6}$$

We use Eq. (9.3.6) together with the initial conditions to get u_i^1 and then go on with Eq. (9.3.3) for $k = 1, 2, \ldots$. Note the following facts:

- We must have $r \le 1$ to guarantee the stability.
- The accuracy of the solution gets better as r becomes larger so that Δx decreases.

It is therefore reasonable to select $r = 1$.

The stability condition can be obtained by substituting Eq. (9.2.4) into Eq. (9.3.3) and applying the Jury test [P-3]:

$$\lambda = 2r\cos(\pi/P) + 2(1 - r) - \lambda^{-1}, \quad \lambda^2 + 2(r(1 - \cos(\pi/P)) - 1)\lambda + 1 = 0$$

We need the solution of this equation to be inside the unit circle for stability, which requires

$$r \le \frac{1}{1-\cos(\pi/P)}, \qquad r = A\frac{\Delta t^2}{\Delta x^2} \le 1 \tag{9.3.7}$$

The objective of the following MATLAB routine "`wave()`" is to implement this algorithm for solving a one-dimensional wave equation.

Example 9.4. A Hyperbolic PDE: One-Dimensional Wave (Vibration). Consider a one-dimensional hyperbolic PDE

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u^2(x,t)}{\partial t^2} \quad \text{for } 0 \le x \le 2,\ 0 \le y \le 2, \text{ and } 0 \le t \le 2 \tag{E9.4.1}$$

with the initial conditions and boundary conditions

$$u(x,0) = x(1-x), \quad \partial u/\partial t(x,0) = 0 \quad \text{for } t = 0 \tag{E9.4.2a}$$

$$u(0,t) = 0 \quad \text{for } x = 0, \quad u(1,t) = 0 \quad \text{for } x = 1 \tag{E9.4.2b}$$

We made the following MATLAB program "`solve_wave.m`" in order to use the routine "`wave()`" to solve this equation and ran this program to get the result shown in Fig. 9.5 and see a dynamic picture.

```
function [u,x,t] = wave(a,xf,T,it0,i1t0,bx0,bxf,M,N)
%solve a u_xx = u_tt  for 0<=x<=xf, 0<=t<=T
% Initial Condition: u(x,0) = it0(x), u_t(x,0) = i1t0(x)
% Boundary Condition: u(0,t)= bx0(t), u(xf,t) = bxf(t)
% M = # of subintervals along x axis
% N = # of subintervals along t axis
dx = xf/M;  x = [0:M]'*dx;
dt = T/N;  t = [0:N]*dt;
for i = 1:M + 1, u(i,1) = it0(x(i)); end
for k = 1:N + 1
   u([1 M + 1],k) = [bx0(t(k)); bxf(t(k))];
end
r = a*(dt/dx)^ 2;  r1 = r/2;  r2 = 2*(1 - r);
u(2:M,2) = r1*u(1:M - 1,1) + (1 - r)*u(2:M,1) + r1*u(3:M + 1,1) ...
                 + dt*i1t0(x(2:M)); %Eq.(9.3.6)
for k = 3:N + 1
  u(2:M,k) = r*u(1:M - 1,k - 1) + r2*u(2:M,k-1) + r*u(3:M + 1,k - 1)...
                 - u(2:M,k - 2); %Eq.(9.3.3)
end
```

```
%solve_wave
a = 1;
it0 = inline('x.*(1-x)','x'); i1t0 = inline('0'); %(E9.4.2a)
bx0t = inline('0');  bxft = inline('0'); %(E9.4.2b)
xf = 1; M = 20;  T = 2; N = 50;
[u,x,t] = wave(a,xf,T,it0,i1t0,bx0t,bxft,M,N);
figure(1), clf
mesh(t,x,u)
figure(2), clf
for n = 1:N %dynamic picture
  plot(x,u(:,n)), axis([0 xf -0.3 0.3]), pause(0.2)
end
```

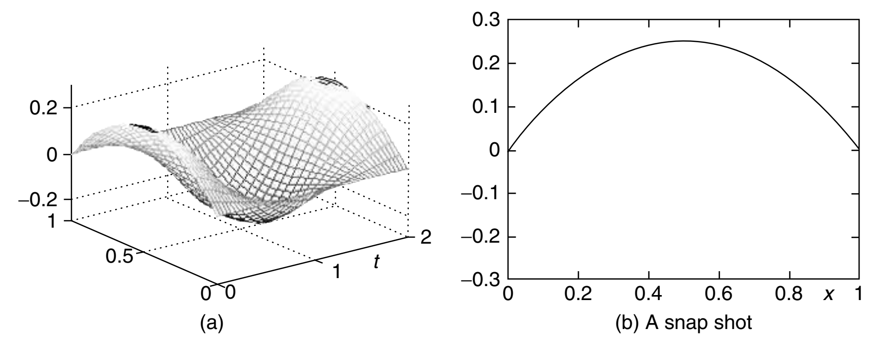


Figure 9.5 A solution for a 1-D hyperbolic PDE obtained by using "`wave()`" (Example 9.4).

9.3.2 Two-Dimensional Hyperbolic PDE

In this section, we consider a two-dimensional wave equation for the amplitude function $u(x, y, t)$ ((x, y) is position, t is time) as

$$A\left(\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2}\right) = \frac{\partial^2 u(x,t)}{\partial t^2} \tag{9.3.8}$$

$$\text{for} \quad 0 \le x \le x_f,\ 0 \le y \le y_f,\ 0 \le t \le T$$

In order for this equation to be solvable, we should be provided with the boundary conditions

$$u(0,y,t) = b_{x0}(y,t), \quad u(x_f,y,t) = b_{xf}(y,t),$$
$$u(x,0,t) = b_{y0}(x,t), \quad \text{and} \quad u(x,y_f,t) = b_{yf}(x,t)$$

as well as the initial condition $u(x, y, 0) = i_0(x, y)$ and $\partial u/\partial t|_{t=0}(x, y, 0) = i_0'(x, y)$.

In the same way as with the one-dimensional case, we replace the second derivatives on both sides by their three-point central difference approximation (5.3.1) as

$$A\left(\frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{\Delta x^2} + \frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{\Delta y^2}\right) = \frac{u_{i,j}^{k+1} - 2u_{i,j}^k + u_{i,j}^{k-1}}{\Delta t^2} \tag{9.3.9}$$

$$\text{with} \quad \Delta x = \frac{x_f}{M_x}, \quad \Delta y = \frac{y_f}{N_y}, \quad \Delta t = \frac{T}{N}$$

which leads to the explicit central difference method:

$$u_{i,j}^{k+1} = r_x(u_{i,j+1}^k + u_{i,j-1}^k) + 2(1 - r_x - r_y)u_{i,j}^k + r_y(u_{i+1,j}^k + u_{i-1,j}^k) - u_{i,j}^{k-1} \tag{9.3.10}$$

$$\text{with} \quad r_x = A\frac{\Delta t^2}{\Delta x^2}, \quad r_y = A\frac{\Delta t^2}{\Delta y^2}$$

Since $u_{i,j}^{-1} = u(x_j, y_i, -\Delta t)$ is not given, we cannot get $u_{i,j}^1$ directly from this formula (9.3.10) with $k = 0$:

$$u_{i,j}^1 = r_x(u_{i,j+1}^0 + u_{i,j-1}^0) + 2(1 - r_x - r_y)u_{i,j}^0 + r_y(u_{i+1,j}^0 + u_{i-1,j}^0) - u_{i,j}^{-1} \tag{9.3.11}$$

Therefore, we approximate the initial condition on the derivative by the central difference as

$$\frac{u_{i,j}^1 - u_{i,j}^{-1}}{2\Delta t} = i_0'(x_j, y_i) \tag{9.3.12}$$

and make use of this to remove $u_{i,j}^{-1}$ from Eq. (9.3.11) to have

$$u_{i,j}^1 = \tfrac{1}{2}\{r_x(u_{i,j+1}^0 + u_{i,j-1}^0) + r_y(u_{i+1,j}^0 + u_{i-1,j}^0)\} + 2(1 - r_x - r_y)u_{i,j}^0 + i_0'(x_j, y_i)\Delta t \tag{9.3.13}$$

We use this Eq. (9.3.13) together with the initial conditions to get $u_{i,j}^1$ and then go on using Eq. (9.3.10) for $k = 1, 2, \ldots$. A sufficient condition for stability [S-1, Section 9.6] is

$$r = \frac{4A\Delta t^2}{\Delta x^2 + \Delta y^2} \le 1 \tag{9.3.14}$$

The objective of the MATLAB routine “`wave2()`” is to implement this algorithm for solving a two-dimensional wave equation.

Example 9.5. A Hyperbolic PDE: Two-Dimensional Wave (Vibration) Over a Square Membrane. Consider a two-dimensional hyperbolic PDE

$$\frac{1}{4}\left(\frac{\partial^2 u(x, y, t)}{\partial x^2} + \frac{\partial^2 u(x, y, t)}{\partial y^2}\right) = \frac{\partial u^2(x, y, t)}{\partial t^2}$$
$$\text{for} \quad 0 \le x \le 2, \quad 0 \le y \le 2 \quad \text{and} \quad 0 \le t \le 2 \tag{E9.5.1}$$

with the zero boundary conditions and the initial conditions

$$u(0, y, t) = 0, \quad u(2, y, t) = 0, \quad u(x, 0, t) = 0, \quad u(x, 2, t) = 0 \tag{E9.5.2}$$

$$u(x, y, 0) = 0.1\sin(\pi x)\sin(\pi y/2), \quad \partial u/\partial t(x, y, 0) = 0 \text{ for } t = 0 \tag{E9.5.3}$$

We made the following MATLAB program “`solve_wave2.m`” in order to use the routine “`wave2()`” for solving this equation and ran this program to get the result shown in Fig. 9.6 and see a dynamic picture. Note that we can be sure of stability, since we have

$$r = \frac{4A\Delta t^2}{\Delta x^2 + \Delta y^2} = \frac{4(1/4)(2/20)^2}{(2/20)^2 + (2/20)^2} = \frac{1}{2} \le 1 \tag{E9.5.4}$$

```
function [u,x,y,t] = wave2(a,D,T,it0,i1t0,bxyt,Mx,My,N)
%solve a(u_xx + u_yy) = u_tt for D(1) <= x <= D(2), D(3) <= y <= D(4), 0 <= t <= T
% Initial Condition: u(x,y,0) = it0(x,y), u_t(x,y,0)  = i1t0(x,y)
% Boundary Condition: u(x,y,t) = bxyt(x,y,t) for (x,y) on Boundary
% Mx/My = # of subintervals along x/y axis
% N = # of subintervals along t axis
dx = (D(2)- D(1))/Mx; x = D(1)+[0:Mx]*dx;
dy = (D(4)- D(3))/My; y = D(3)+[0:My]'*dy;
dt = T/N;  t = [0:N]*dt;
%Initialization
u = zeros(My+1,Mx + 1);  ut = zeros(My + 1,Mx + 1);
for j = 2:Mx
  for i = 2:My
    u(i,j) = it0(x(j),y(i));  ut(i,j) = i1t0(x(j),y(i));
  end
end
adt2 = a*dt*dt; rx = adt2/(dx*dx); ry = adt2/(dy*dy);
rxy1 = 1- rx - ry; rxy2 = rxy1*2;
u_1 = u;
for k = 0:N
  t = k*dt;
  for i = 1:My + 1 %Boundary condition
    u(i,[1 Mx + 1]) = [bxyt(x(1),y(i),t) bxyt(x(Mx + 1),y(i),t)];
  end
  for j = 1:Mx + 1
    u([1 My + 1],j) = [bxyt(x(j),y(1),t); bxyt(x(j),y(My + 1),t)];
  end
  if k = = 0
    for i = 2:My
      for j = 2:Mx  %Eq.(9.3.13)
        u(i,j) = 0.5*(rx*(u_1(i,j - 1) + u_1(i,j + 1))...
            + ry*(u_1(i - 1,j)+u_1(i + 1,j))) + rxy1*u(i,j) + dt*ut(i,j);
      end
    end
   else
    for i = 2:My
      for j = 2:Mx  %Eq.(<eqnr>9.3.10)</eqnr>
        u(i,j) = rx*(u_1(i,j - 1)+ u_1(i,j + 1))...
           + ry*(u_1(i - 1,j) + u_1(i + 1,j)) + rxy2*u(i,j) -u_2(i,j);
      end
    end
  end
  u_2 = u_1;  u_1 = u; %update the buffer memory
  mesh(x,y,u), axis([0 2 0 2 -.1 .1]), pause
end
```

```
%solve_wave2
it0 = inline('0.1*sin(pi*x)*sin(pi*y/2)','x','y'); %(E9.5.3)
i1t0 = inline('0','x','y');  bxyt = inline('0','x','y','t'); %(E9.5.2)
a = .25; D = [0 2 0 2]; T = 2; Mx = 40; My = 40; N = 40;
[u,x,y,t] = wave2(a,xf,T,it0,i1t0,bxyt,Mx,My,N);
```

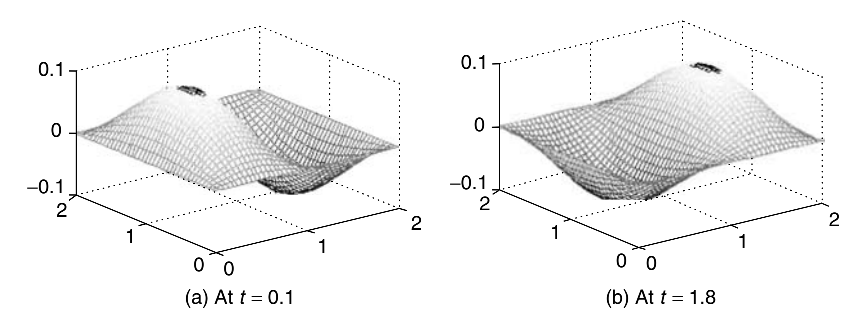


Figure 9.6 The solution of a two-dimensional hyperbolic PDE: vibration of a square membrane (Example 9.5).

9.4 FINITE ELEMENT METHOD (FEM) FOR SOLVING PDE

The FEM method is another procedure used in finding approximate numerical solutions to BVPs/PDEs. It can handle irregular boundaries in the same way as regular boundaries [R-1, S-2, Z-1]. It consists of the following steps to solve the elliptic PDE:

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} + g(x,y)u(x,y) = f(x,y) \tag{9.4.1}$$

for the domain D enclosed by the boundary B on which the boundary condition is given as

$$u(x,y) = b(x,y) \qquad \text{on the boundary } B \tag{9.4.2}$$

1. Discretize the (two-dimensional) domain D into, say, N_s subregions $\{S_1, S_2, \ldots, S_{Ns}\}$ such as triangular elements, neither necessarily of the same size nor necessarily covering the entire domain completely and exactly.
2. Specify the positions of N_n nodes and number them starting from the boundary nodes, say, $n = 1, \ldots, N_b$, and then the interior nodes, say, $n = N_b + 1, \ldots, N_n$.
3. Define the basis/shape/interpolation functions

$$\phi_n(x,y) = \{\phi_{n,s}, \text{for } s = 1, \ldots, N_s\} \;\forall\; (x,y) \in D \tag{9.4.3a}$$

$$\phi_{n,s}(x,y) = p_{n,s}(1) + p_{n,s}(2)x + p_{n,s}(3)y \quad \text{for each subregion } S_s \tag{9.4.3b}$$

collectively for all subregions $s = 1 : N_s$ and for each node $n = 1 : N_n$, so that ϕ_n is 1 only at node n, and 0 at all other nodes. Then, the approximate solution of the PDE is a linear combination of basis functions $\phi_n(x,y)$ as

$$u(x,y) = \mathbf{c}^T \boldsymbol{\varphi}(x,y) = \sum_{n=1}^{N_n} c_n \phi_n(x,y) = \sum_{n=1}^{N_b} c_n \phi_n + \sum_{n=N_b+1}^{N_n} c_n \phi_n = \mathbf{c}_1^T \boldsymbol{\varphi}_1 + \mathbf{c}_2^T \boldsymbol{\varphi}_2 \tag{9.4.4}$$

where

$$\boldsymbol{\varphi}_1 = [\,\phi_1 \quad \phi_2 \quad \cdot \quad \phi_{N_b}\,]^T, \quad \mathbf{c}_1 = [\,c_1 \quad c_2 \quad \cdot \quad c_{N_b}\,]^T \tag{9.4.5a}$$

$$\boldsymbol{\varphi}_2 = [\,\phi_{N_b+1} \quad \phi_{N_b+2} \quad \cdot \quad \phi_{N_n}\,]^T, \quad \mathbf{c}_2 = [\,c_{N_b+1} \quad c_{N_b+2} \quad \cdot \quad c_{N_n}\,]^T \tag{9.4.5b}$$

For each subregion $s = 1, \ldots, N_s$, this solution can be written as

$$\phi_s(x, y) = \sum_{n=1}^{N_n} c_n \phi_{n,s}(x, y) = \sum_{n=1}^{N_n} c_n (p_{n,s}(1) + p_{n,s}(2)x + p_{n,s}(3)y) \tag{9.4.6}$$

4. Set the values of the boundary node coefficients in $\mathbf{c}_1$ to the boundary values according to the boundary condition.
5. Determine the values of the interior node coefficients in $\mathbf{c}_2$ by solving the system of equations

$$A_2 \mathbf{c}_2 = \mathbf{d} \tag{9.4.7}$$

where

$$A_1 = \sum_{s=1}^{N_s} \left\{ \left[\frac{\partial}{\partial x}\boldsymbol{\varphi}_{2,s}\right]\left[\frac{\partial}{\partial x}\boldsymbol{\varphi}_{1,s}\right]^T + \left[\frac{\partial}{\partial y}\boldsymbol{\varphi}_{2,s}\right]\left[\frac{\partial}{\partial y}\boldsymbol{\varphi}_{1,s}\right]^T - g(x_s, y_s)\boldsymbol{\varphi}_{2,s}\boldsymbol{\varphi}_{1,s}^T \right\} \Delta S_s \tag{9.4.8}$$

$$\boldsymbol{\varphi}_{1,s} = [\,\phi_{1,s} \quad \phi_{2,s} \quad \cdot \quad \phi_{N_b,s}\,]^T$$

$$\frac{\partial}{\partial x}\boldsymbol{\varphi}_{1,s} = [\,p_{1,s}(2) \quad p_{2,s}(2) \quad \cdot \quad p_{N_b,s}(2)\,]^T$$

$$\frac{\partial}{\partial y}\boldsymbol{\varphi}_{1,s} = [\,p_{1,s}(3) \quad p_{2,s}(3) \quad \cdot \quad p_{Nb,s}(3)\,]^T$$

$$A_2 = \sum_{s=1}^{N_s} \left\{ \left[\frac{\partial}{\partial x}\boldsymbol{\varphi}_{2,s}\right]\left[\frac{\partial}{\partial x}\boldsymbol{\varphi}_{2,s}\right]^T + \left[\frac{\partial}{\partial y}\boldsymbol{\varphi}_{2,s}\right]\left[\frac{\partial}{\partial y}\boldsymbol{\varphi}_{2,s}\right]^T - g(x_s, y_s)\boldsymbol{\varphi}_{2,s}\boldsymbol{\varphi}_{2,s}^T \right\} \Delta S_s \tag{9.4.9}$$

$$\boldsymbol{\varphi}_{2,s} = [\,\phi_{Nb+1,s} \quad \phi_{Nb+2,s} \quad \cdot \quad \phi_{Nn,s}\,]^T$$

$$\frac{\partial}{\partial x}\boldsymbol{\varphi}_{2,s} = [\,p_{Nb+1,s}(2) \quad \phi_{Nb+2,s}(2) \quad \cdot \quad \phi_{Nn,s}(2)\,]^T$$

$$\frac{\partial}{\partial y}\boldsymbol{\varphi}_{2,s} = [\,p_{Nb+1,s}(3) \quad \phi_{Nb+2,s}(3) \quad \cdot \quad \phi_{Nn,s}(3)\,]^T$$

$$\mathbf{d} = -A_1\mathbf{c}_1 - \sum_{s=1}^{N_s} f(x_s, y_s)\boldsymbol{\varphi}_{2,s}\Delta S \tag{9.4.10}$$

(x_s, y_s): the centroid (gravity center) of the sth subregion S_s

The FEM is based on the variational principle that a solution to Eq. (9.4.1) can be obtained by minimizing the functional

$$I = \iint_R \left\{ \left(\frac{\partial}{\partial x}u(x, y)\right)^2 + \left(\frac{\partial}{\partial y}u(x, y)\right)^2 - g(x, y)u^2(x, y) + 2f(x, y)u(x, y) \right\} dx\,dy \tag{9.4.11}$$

which, with $u(x, y) = \mathbf{c}^T \boldsymbol{\varphi}(x, y)$, can be written as

$$I = \iint_R \left\{ \mathbf{c}^T \frac{\partial}{\partial x}\boldsymbol{\varphi} \frac{\partial}{\partial x}\boldsymbol{\varphi}^T \mathbf{c} + \mathbf{c}^T \frac{\partial}{\partial y}\boldsymbol{\varphi} \frac{\partial}{\partial y}\boldsymbol{\varphi}^T \mathbf{c} - g(x, y)\mathbf{c}^T \boldsymbol{\varphi}\boldsymbol{\varphi}^T \mathbf{c} + 2f(x, y)\mathbf{c}^T \boldsymbol{\varphi} \right\} dx\, dy \tag{9.4.12}$$

The condition for this functional to be minimized with respect to $\mathbf{c}$ is

$$\frac{d}{d\mathbf{c}_2} I = \iint_R \left\{ \frac{\partial}{\partial x}\boldsymbol{\varphi}_2 \frac{\partial}{\partial x}\boldsymbol{\varphi}^T \mathbf{c} + \frac{\partial}{\partial y}\boldsymbol{\varphi}_2 \frac{\partial}{\partial y}\boldsymbol{\varphi}^T \mathbf{c} - g(x, y)\boldsymbol{\varphi}_2 \boldsymbol{\varphi}^T \mathbf{c} + f(x, y)\boldsymbol{\varphi}_2 \right\} dx\, dy = 0 \tag{9.4.13}$$

$$\approx A_1 \mathbf{c}_1 + A_2 \mathbf{c}_2 + \sum_{s=1}^{N_s} f(x_s, y_s)\boldsymbol{\varphi}_{2,s} \Delta S_s = 0 \tag{9.4.14}$$

See [R-1] for details.

The objectives of the MATLAB routines "`fem_basis_ftn()`" and "`fem_coef()`" are to construct the basis function $\phi_{n,s}(x, y)$'s for each node $n = 1, \ldots, N_n$ and each subregion $s = 1, \ldots, N_s$ and to get the coefficient vector $\mathbf{c}$ of the solution (9.4.4) via Eq. (9.4.7) and the solution polynomial $\phi_s(x, y)$'s via Eq. (9.4.6) for each subregion $s = 1, \ldots, N_s$, respectively.

Before going into a specific example of applying the FEM method to solve a PDE, let us take a look at the basis (shape) function $\phi_n(x, y)$ for each node $n = 1, \ldots, N_n$, which is defined collectively for all of the (triangular) subregions so that ϕ_n is 1 only at node n, and 0 at all other nodes and can be generated by the routine "`fem_basis_ftn()`".

```
function p = fem_basis_ftn(N,S)
%p(i,s,1:3): coefficients of each basis ftn phi_i
%               for s-th subregion(triangle)
%N(n,1:2) : x & y coordinates of the n-th node
%S(s,1:3) : the node #s of the s-th subregion(triangle)
N_n = size(N,1); % the total number of nodes
N_s = size(S,1); % the total number of subregions(triangles)
for n = 1:N_n
  for s = 1:N_s
    for i = 1:3
      A(i,1:3) = [1 N(S(s,i),1:2)];
      b(i) = (S(s,i) == n); %The nth basis ftn is 1 only at node n.
    end
    pnt=A\b';
    for i=1:3, p(n,s,i) = pnt(i); end
  end
end
```

```
function [U,c] = fem_coef(f,g,p,c,N,S,N_i)
%p(i,s,1:3): coefficients of basis ftn phi_i for the s-th subregion
%c = [ .1 1 . 0 0 .] with value for boundary and 0 for interior nodes
%N(n,1:2) : x & y coordinates of the n-th node
%S(s,1:3) : the node #s of the s-th subregion(triangle)
%N_i      : the number of the interior nodes
%U(s,1:3) : the coefficients of p1 + p2(s)x + p3(s)y for each subregion
N_n = size(N,1); % the total number of nodes = N_b + N_i
N_s = size(S,1); % the total number of subregions(triangles)
d=zeros(N_i,1);
N_b = N_n-N_i;
for i = N_b+1:N_n
  for n = 1:N_n
    for s = 1:N_s
      xy = (N(S(s,1),:) + N(S(s,2),:) + N(S(s,3),:))/3; %gravity center
       %phi_i,x*phi_n,x + phi_i,y*phi_n,y - g(x,y)*phi_i*phi_n
      p_vctr = [p([i n],s,1) p([i n],s,2) p([i n],s,3)];
      tmpg(s) = sum(p(i,s,2:3).*p(n,s,2:3))...
           -g(xy(1),xy(2))*p_vctr(1,:)*[1 xy]'*p_vctr(2,:)*[1 xy]';
      dS(s) = det([N(S(s,1),:) 1; N(S(s,2),:) 1;N(S(s,3),:) 1])/2;
       %area of triangular subregion
      if n == 1, tmpf(s) = -f(xy(1),xy(2))*p_vctr(1,:)*[1 xy]'; end
    end
    A12(i - N_b,n) = tmpg*abs(dS)'; %Eqs. (9.4.8),(9.4.9)
  end
  d(i-N_b) = tmpf*abs(dS)'; %Eq.(9.4.10)
end
d = d - A12(1:N_i,1:N_b)*c(1:N_b)'; %Eq.(9.4.10)
c(N_b + 1:N_n) = A12(1:N_i,N_b+1:N_n)\d; %Eq.(9.4.7)
for s = 1:N_s
  for j = 1:3, U(s,j) = c*p(:,s,j); end %Eq.(9.4.6)
end
```

Actually, we will plot the basis (shape) functions for the region divided into four triangular subregions as depicted in Fig. 9.7 in two ways. First, we generate the basis functions by using the routine "`fem_basis_ftn()`" and plot one of them for node 1 by using the MATLAB command `mesh()`, as depicted in Fig. 9.8a. Second, without generating the basis functions, we use the MATLAB command "`trimesh()`" to plot the shape functions for nodes $n = 2$, 3, 4, and 5 as depicted in Figs. 9.8b–e, each of which is 1 only at the corresponding node n and is 0 at all other nodes. Figure 9.8f is the graph of a linear combination of basis functions

$$u(x, y) = \mathbf{c}^T \boldsymbol{\varphi}(x, y) = \sum_{n=1}^{N_n} c_n \phi_n(x, y) \tag{9.4.15}$$

having the given value c_n at each node n. This can obtained by using the MATLAB command "`trimesh()`" as

```
>>trimesh(S,N(:,1),N(:,2),c)
```

where the first input argument `S` has the node numbers for each subregion, the second/third input argument `N` has the x/y coordinates for each node, and the

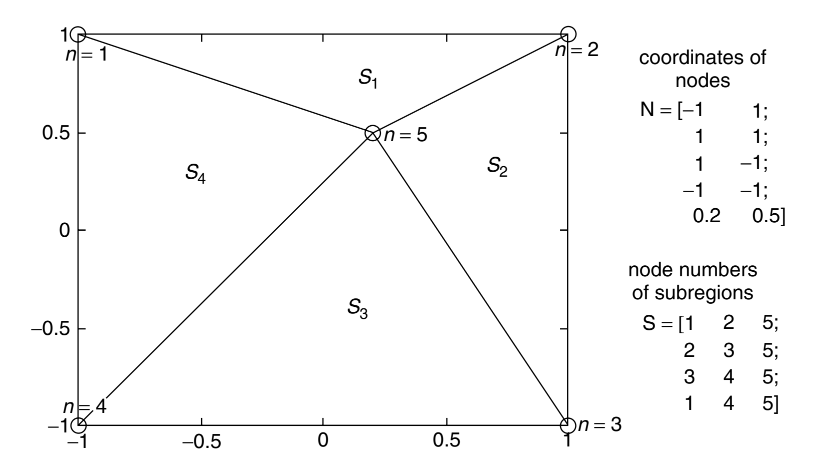


Figure 9.7 A region (domain) divided into four triangular subregions.

fourth input argument c has the function values at each node as follows:

$$S = \begin{bmatrix} 1 & 2 & 5 \\ 2 & 3 & 5 \\ 3 & 4 & 5 \\ 1 & 4 & 5 \end{bmatrix}, \quad N = \begin{bmatrix} -1 & 1 \\ 1 & 1 \\ 1 & -1 \\ -1 & -1 \\ 0.2 & 0.5 \end{bmatrix}, \quad c = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 0 \end{bmatrix} \tag{9.4.16}$$

For this job, we make the following program "`show_basis.m`" and run it to get Figs. 9.7 and 9.8 together with the coefficients of each basis function as

$$p(:,:,1) = \begin{bmatrix} -3/10 & 0 & 0 & -1/8 \\ -7/10 & -7/16 & 0 & 0 \\ 0 & 3/16 & 1/10 & 0 \\ 0 & 0 & 7/30 & 7/24 \\ 2 & 5/4 & 2/3 & 5/6 \end{bmatrix},$$

$$p(:,:,2) = \begin{bmatrix} -1/2 & 0 & 0 & -5/8 \\ 1/2 & 15/16 & 0 & 0 \\ 0 & 5/16 & 1/2 & 0 \\ 0 & 0 & -1/2 & -5/24 \\ 0 & -5/4 & 0 & 5/6 \end{bmatrix}, \tag{9.4.17}$$

$$p(:,:,3) = \begin{bmatrix} 4/5 & 0 & 0 & 1/2 \\ 6/5 & 1/2 & 0 & 0 \\ 0 & -1/2 & -2/5 & 0 \\ 0 & 0 & -4/15 & -1/2 \\ -2 & 0 & 2/3 & 0 \end{bmatrix}$$

The meaning of this N_n (the number of nodes:5) $\times$ N_s (the number of subregions:4) $\times$ 3 array p is that, say, the second rows of the three sub-arrays constitute

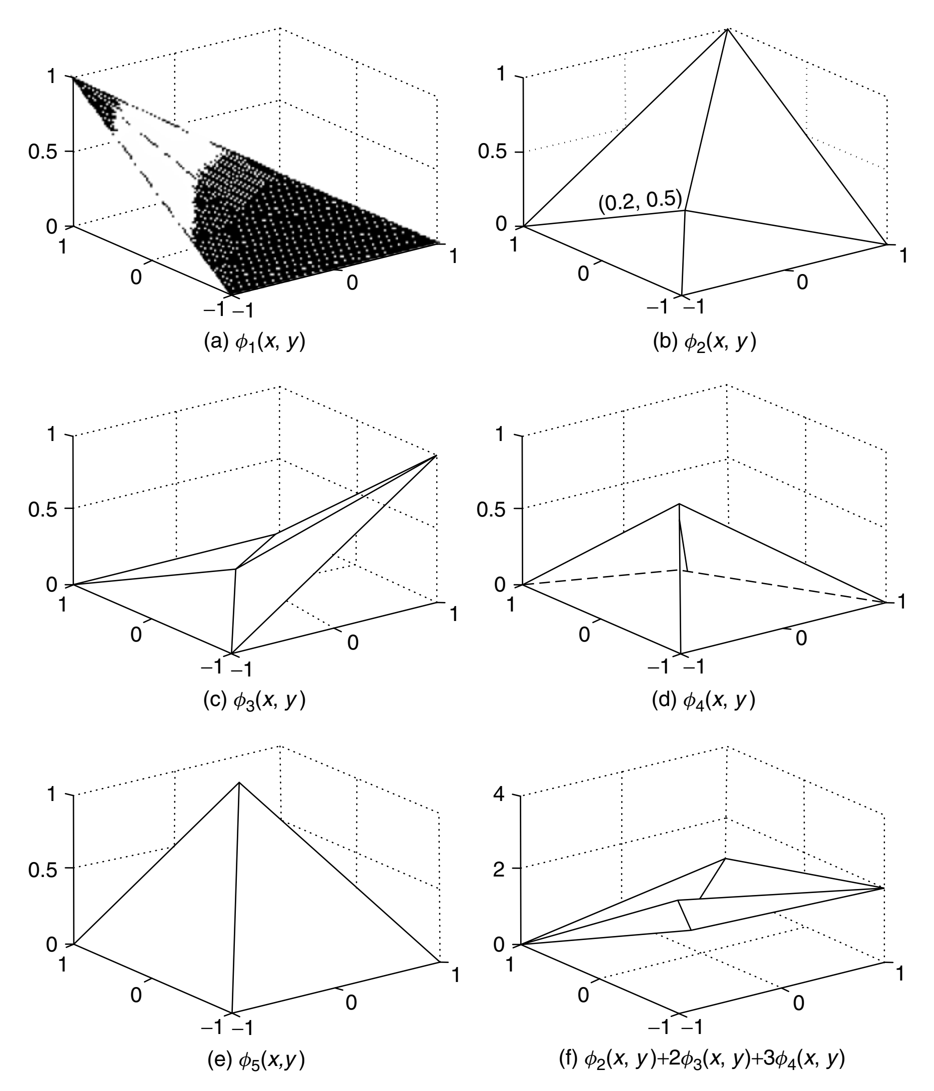


Figure 9.8 The basis (shape) functions for nodes in Fig. 9.7 and a composite function.

the coefficient vectors of the basis function for node 2 as

$$\phi_2(x, y) = \begin{cases} -7/10 + (1/2)x + (6/5)y & \text{for subregion } S_1 \\ -7/16 + (15/16)x + (1/2)y & \text{for subregion } S_2 \\ 0 + 0 \cdot x + 0 \cdot y & \text{for subregion } S_3 \\ 0 + 0 \cdot x + 0 \cdot y & \text{for subregion } S_4 \end{cases} \tag{9.4.18}$$

which turns out to be 1 only at node 2 [i.e., (1,1)] and 0 at all other nodes and on the subregions that do not have node 2 as their vertex, as depicted in Fig. 9.8b.

With the program "show_basis.m" in your computer, type the following commands into the MATLAB command window and see the graphical/textual output.

```
>>show_basis
>>p
```

Now, let us see the following example.

```
%show_basis
clear
N = [-1 1;1 1;1 -1;-1 -1;0.2 0.5]; %the list of nodes in Fig.9.7
N_n = size(N,1); % the number of nodes
S = [1 2 5;2 3 5;3 4 5;1 4 5]; %the list of subregions in Fig.9.7
N_s = size(S,1); % the number of subregions
figure(1), clf
for s = 1:N_s
  nodes = [S(s,:) S(s,1)];
  for i = 1:3
    plot([N(nodes(i),1) N(nodes(i + 1),1)], ...
     [N(nodes(i),2) N(nodes(i+1),2)]), hold on
   end
end
%basis/shape function
p = fem_basis_ftn(N,S);
x0 = -1; xf = 1; y0 = -1; yf = 1; %graphic region
figure(2), clf
Mx = 50; My = 50;
dx = (xf - x0)/Mx; dy = (yf - y0)/My;
xi = x0 + [0:Mx]*dx; yi = y0 + [0:My]*dy;
i_ns = [1 2 3 4 5]; %the list of node numbers whose basis ftn to plot
for itr = 1:5
  i_n = i_ns(itr);
  if itr == 1
    for i = 1:length(xi)
      for j = 1:length(yi)
        Z(j,i) = 0;
        for s = 1:N_s
          if inpolygon(xi(i),yi(j), N(S(s,:),1),N(S(s,:),2)) > 0
            Z(j,i) = p(i_n,s,1) + p(i_n,s,2)*xi(i) + p(i_n,s,3)*yi(j);
            break;
          end
        end
      end
    end
    subplot(321), mesh(xi,yi,Z) %basis function for node 1
   else
    c1 = zeros(size(c)); c1(i_n) = 1;
    subplot(320 + itr)
    trimesh(S,N(:,1),N(:,2),c1) %basis function for node 2-5
  end
end
c = [0 1 2 3 0]; %the values for all nodes
subplot(326)
trimesh(S,N(:,1),N(:,2),c) %Fig.9.8f: a composite function
```

Example 9.6. Laplace's Equation: Electric Potential Over a Plate with Point Charge. Consider the following Laplace's equation:

$$\nabla^2 u(x, y) = \frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = f(x, y) \qquad \text{(E9.6.1)}$$

$$\text{for } -1 \le x \le +1, -1 \le y \le +1$$

where

$$f(x, y) = \begin{cases} -1 & \text{for } (x, y) = (0.5, 0.5) \\ +1 & \text{for } (x, y) = (-0.5, -0.5) \\ 0 & \text{elsewhere} \end{cases} \qquad \text{(E9.6.2)}$$

and the boundary condition is $u(x, y) = 0$ for all boundaries of the rectangular domain.

In order to solve this equation by using the FEM, we locate 12 boundary points and 19 interior points, number them, and divide the domain into 36 triangular subregions as depicted in Fig. 9.9. Note that we have made the size of the subregions small and their density high around the points (+0.5, +0.5) and

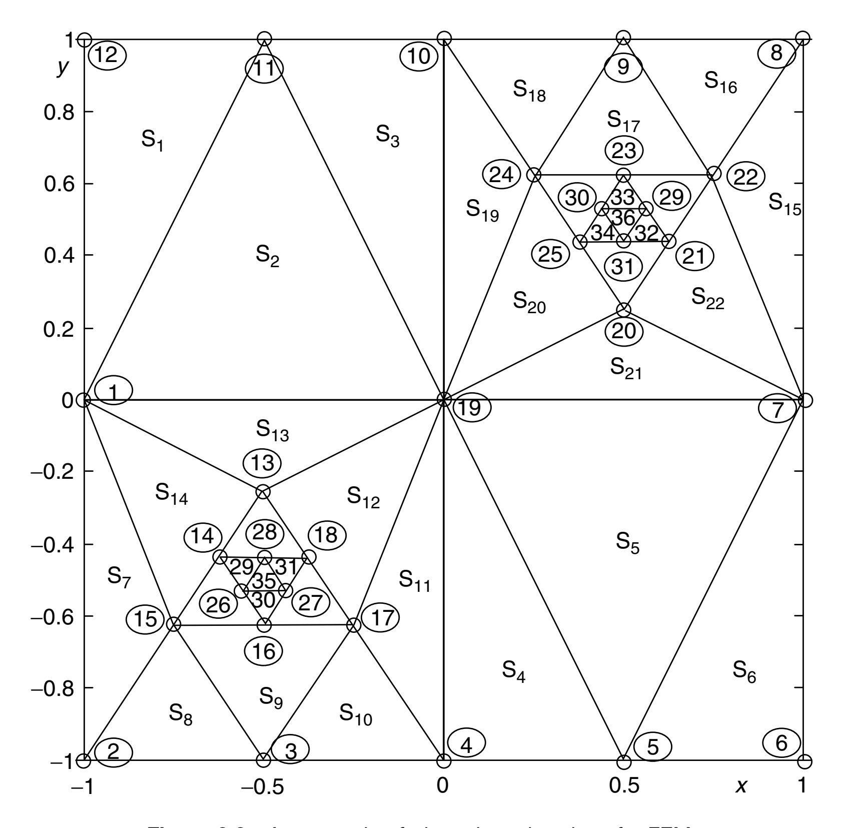
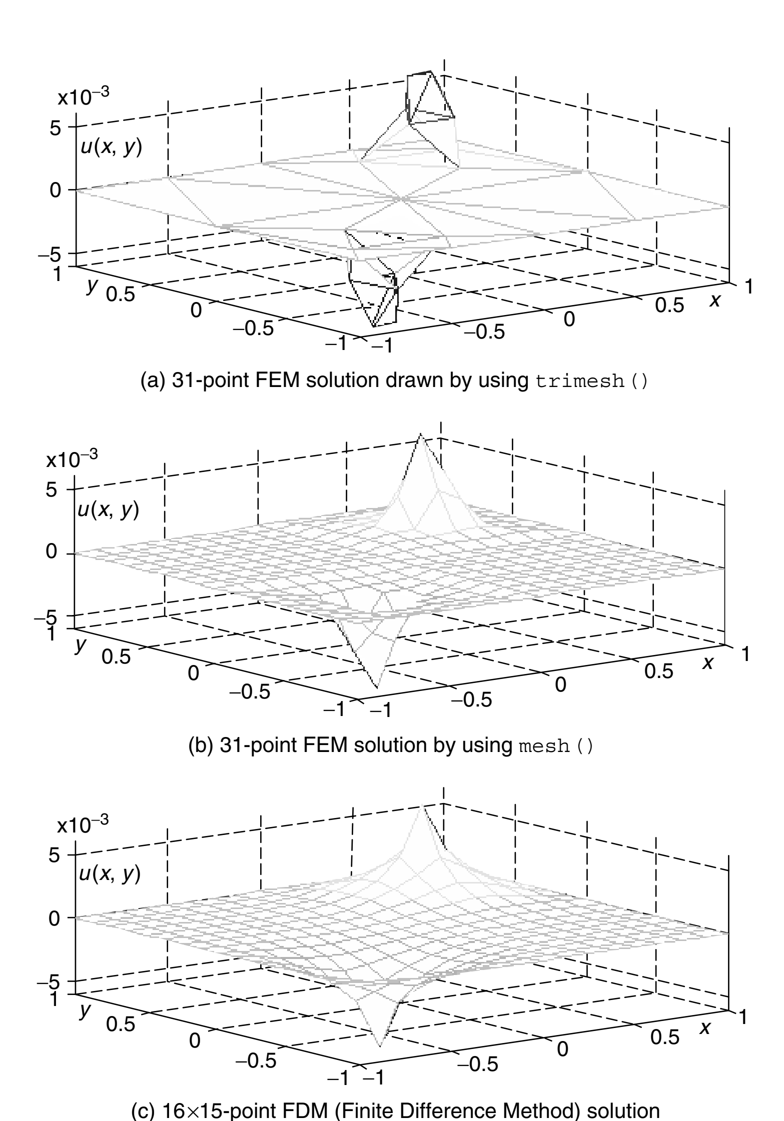


Figure 9.9 An example of triangular subregions for FEM.

(−0.5, −0.5), since they are only two points at which the value of the right-hand side of Eq. (9.6.1) is not zero, and consequently the value of the solution $u(x, y)$ is expected to change sensitively around them.

We made the following MATLAB program "`do_fem.m`" in order to use the routines "`fem_basis_ftn()`" and "`fem_coef()`" for solving this equation. For comparison, we have added the statements to solve the same equation by using the routine "`poisson()`" (Section 9.1). The results obtained by running this program are depicted in Fig. 9.10a–c.



(a) 31-point FEM solution drawn by using `trimesh()`

(b) 31-point FEM solution by using `mesh()`

(c) 16×15-point FDM (Finite Difference Method) solution

Figure 9.10 Results of Example 9.6.

```
%do_fem
% for Example 9.6
clear
N = [-1 0;-1 -1;-1/2 -1;0 -1;1/2 -1; 1 -1;1 0;1 1;1/2 1; 0 1;
     -1/2 1;-1 1; -1/2 -1/4; -5/8 -7/16;-3/4 -5/8;-1/2 -5/8;
     -1/4 -5/8;-3/8 -7/16; 0  0; 1/2 1/4;5/8 7/16;3/4 5/8;
      1/2 5/8;1/4 5/8;3/8 7/16;-9/16 -17/32;-7/16 -17/32;
     -1/2 -7/16;9/16 17/32;7/16 17/32;1/2 7/16]; %nodes
N_b = 12; %the number of boundary nodes
S = [1 11 12;1 11 19;10 11 19;4 5 19;5 7 19; 5 6 7;1 2 15; 2 3 15;
     3 15 17;3 4 17;4 17 19;13 17 19;1 13 19;1 13 15;7 8 22;8 9 22;
     9 22 24;9 10 24; 10 19 24; 19 20 24;7 19 20; 7 20 22;13 14 18;
     14 15 16;16 17 18;20 21 25;21 22 23;23 24 25;14 26 28;
     16 26 27;18 27 28; 21 29 31;23 29 30;25 30 31;
     26 27 28; 29 30 31]; %triangular subregions
f962 = '(norm([x y]+[0.5 0.5])<0.01)-(norm([x y]-[0.5 0.5]) < 0.01)';
f=inline(f962,'x','y'); %(E9.6.2)
g=inline('0','x','y');
N_n = size(N,1); %the total number of nodes
N_i = N_n - N_b; %the number of interior nodes
c = zeros(1,N_n); %boundary value or 0 for boundary/interior nodes
p = fem_basis_ftn(N,S);
[U,c] = fem_coef(f,g,p,c,N,S,N_i);
%Output through the triangular mesh-type graph
figure(1), clf, trimesh(S,N(:,1),N(:,2),c)
%Output through the rectangular mesh-type graph
N_s = size(S,1); %the total number of subregions(triangles)
x0 = -1; xf = 1; y0 = -1; yf = 1;
Mx = 16; dx = (xf - x0)/Mx; xi = x0+[0:Mx]*dx;
My = 16; dy = (yf - y0)/My; yi = y0+[0:My]*dy;
for i = 1:length(xi)
  for j = 1:length(yi)
    for s = 1:N_s %which subregion the point belongs to
      if inpolygon(xi(i),yi(j), N(S(s,:),1),N(S(s,:),2)) > 0
        Z(i,j) = U(s,:)*[1 xi(i) yi(j)]'; %Eq.(9.4.5b)
        break;
      end
    end
  end
end
figure(2), clf, mesh(xi,yi,Z)
%For comparison
bx0 = inline('0');  bxf = inline('0');
by0 = inline('0');  byf = inline('0');
[U,x,y] = poisson(f,g,bx0,bxf,by0,byf,[x0 xf y0 yf],Mx,My);
figure(3), clf, mesh(x,y,U)
```

9.5 GUI OF MATLAB FOR SOLVING PDES: PDETOOL

In this section, we will see what problems can be solved by using the GUI (graphic user interface) tool of MATLAB for PDEs and then apply the tool to solve the elliptic/parabolic/hyperbolic equations dealt with in Examples 9.1/9.3/9.5 and 9.6.

9.5.1 Basic PDEs Solvable by PDETOOL

Basically, the PDE toolbox can be used for the following kinds of PDE.

1. Elliptic PDE

$$-\nabla \cdot (c\nabla u) + au = f \quad \text{over a domain } \Omega \tag{9.5.1}$$

with some boundary conditions like

$$\begin{aligned} hu &= r \ \text{(Dirichlet condition)} \\ \text{or} \quad \mathbf{n} \cdot c\nabla u + qu &= g \ \text{(generalized Neumann condition)} \end{aligned} \tag{9.5.2}$$

on the boundary $\partial\Omega$, where $\mathbf{n}$ is the outward unit normal vector to the boundary.

Note that, in case u is a scalar-valued function on a rectangular domain as depicted in Fig. 9.1, Eq. (9.5.1) becomes

$$-c\left(\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2}\right) + au(x,y) = f(x,y) \tag{9.5.3}$$

and if the boundary condition for the left-side boundary segment is of Neumann type like Eq. (9.1.7), Eq. (9.5.2) can be written as

$$\begin{aligned} &-\mathbf{i} \cdot c\left(\frac{\partial u(x,y)}{\partial x}\mathbf{i} + \frac{\partial u(x,y)}{\partial y}\mathbf{j}\right) + qu(x,y) \\ &= -c\frac{\partial u(x,y)}{\partial x} + qu(x,y) = g(x,y) \end{aligned} \tag{9.5.4}$$

since the outward unit normal vector to the left-side boundary is $\mathbf{n} = \mathbf{i}$, where $\mathbf{i}$ and $\mathbf{j}$ are the unit vectors along the x axis and y-axis, respectively.

2. Parabolic PDE

$$-\nabla \cdot (c\nabla u) + au + d\frac{\partial u}{\partial t} = f \tag{9.5.5}$$

over a domain Ω and for a time range $0 \le t \le T$

with boundary conditions like Eq. (9.5.2) and, additionally, the initial condition $u(t_0)$.

3. Hyperbolic PDE

$$-\nabla \cdot (c\nabla u) + au + d\frac{\partial^2 u}{\partial t^2} = f \tag{9.5.6}$$

over a domain Ω and for a time range $0 \le t \le T$

with boundary conditions like Eq. (9.5.2) and, additionally, the initial conditions $u(t_0)/u'(t_0)$.

4. Eigenmode PDE

$$-\nabla \cdot (c\nabla u) + au = \lambda du \tag{9.5.7}$$

over a domain Ω and for an unknown eigenvalue λ

with some boundary conditions like Eq. (9.5.2).

The PDE toolbox can also deal with a system of PDEs like

$$\begin{aligned} -\nabla \cdot (c_{11}\nabla u_1) - \nabla \cdot (c_{12}\nabla u_2) + a_{11}u_1 + a_{12}u_2 &= f_1 \\ -\nabla \cdot (c_{21}\nabla u_1) - \nabla \cdot (c_{22}\nabla u_2) + a_{21}u_1 + a_{22}u_2 &= f_2 \end{aligned} \quad \text{over a domain } \Omega \tag{9.5.8}$$

with Dirichlet boundary conditions like

$$\begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} \tag{9.5.9}$$

or generalized Neumann boundary conditions like

$$\begin{aligned} \mathbf{n} \cdot (c_{11}\nabla u_1) + \mathbf{n} \cdot (c_{12}\nabla u_2) + q_{11}u_1 + q_{12}u_2 &= g_1 \\ \mathbf{n} \cdot (c_{21}\nabla u_1) + \mathbf{n} \cdot (c_{22}\nabla u_2) + q_{21}u_1 + q_{22}u_2 &= g_2 \end{aligned} \tag{9.5.10}$$

or mixed boundary conditions, where

$$c = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}, \quad a = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad f = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}, \quad u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$h = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}, \quad r = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}, \quad q = \begin{bmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix}, \quad g = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}$$

9.5.2 The Usage of PDETOOL

The PDEtool in MATLAB solves PDEs by using the FEM (finite element method). We should take the following steps to use it.

0. Type '`pdetool` ' into the MATLAB command window to have the PDE toolbox window on the screen as depicted in Fig. 9.11. You can toggle on/off the grid by clicking 'Grid' in the Options pull-down menu (Fig. 9.12a). You can also adjust the ranges of the x axis and the y axis in the box window opened by clicking 'Axes_Limits' in the Options pull-down menu. If you want the rectangles to be aligned with the grid lines, click 'Snap(-to-grid)' in the Options pull-down menu (Fig. 9.12a). If you

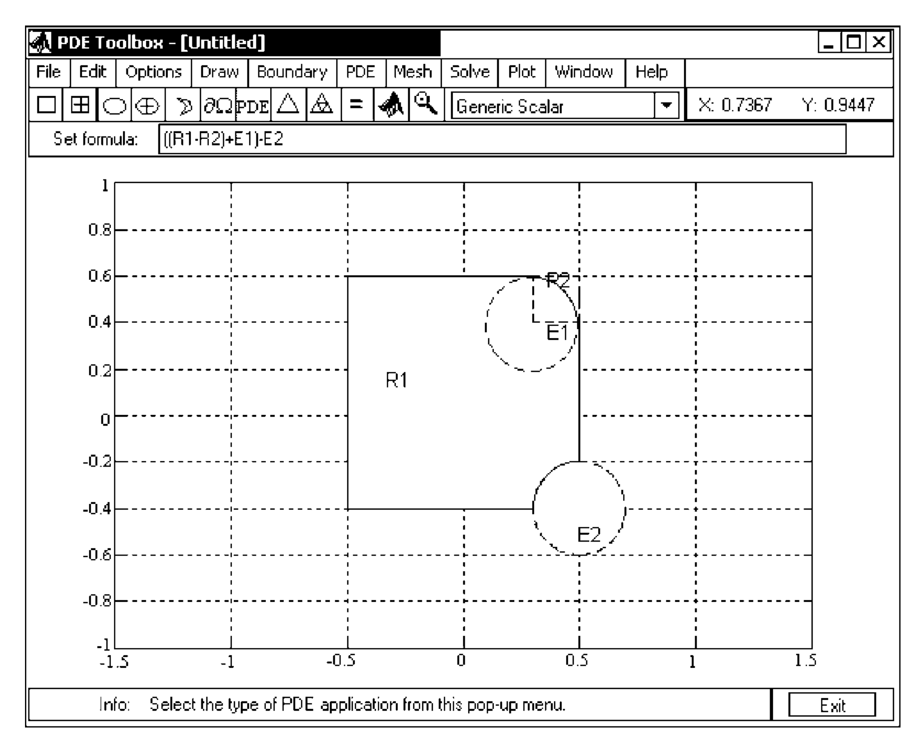


Figure 9.11 The GUI (graphical user interface) window of the MATLAB PDEtool.

want to have the x axis and the y axis of equal scale so that a circle/square may not look like an ellipse/rectangle, click 'Axes_Equal' in the Options pull-down menu. You can choose the type of PDE problem you want to solve in the submenu popped out by clicking 'Application' in the Options pull-down menu (Fig. 9.12a).

(cf) In order to be able to specify the boundary condition for a boundary segment by clicking it, the segment must be inside in the graphic region of PDEtool.

1. In Draw mode, you can create the two-dimensional geometry of domain Ω by using the constructive solid geometry (CSG) paradigm, which enables us to make a set of solid objects such as rectangles, circles/ellipses, and polygons. In order to do so, click the object that you want to draw in the Draw pull-down menu (Fig. 9.12b) or click the button with the corresponding icon (□, ⊞, ⬭, ⊕, ⋗) in the tool-bar just below the top menu-bar (Fig. 9.11). Then, you can click-and-drag to create/move the object of any size at any position as you like. Once an object is drawn, it can be selected by clicking on it. Note that the selected object becomes surrounded by a black solid line and can be deleted by pressing Delete or ^R(Ctrl-R) key. The created object is automatically labeled, but it can be relabeled and resized (numerically) through the Object dialog box opened

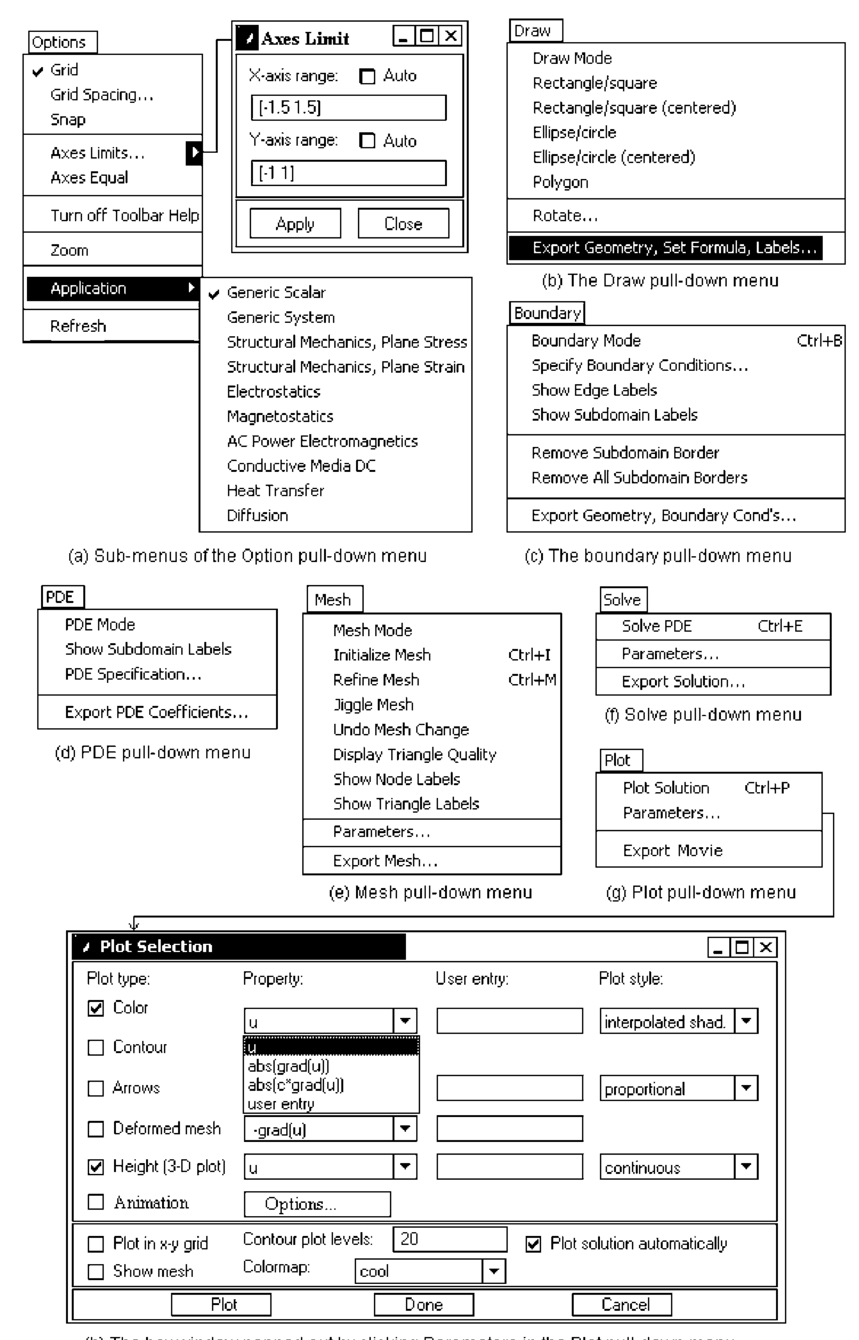


Figure 9.12 Pull-down menu from the top menu and its submenu of the MATLAB PDEtool.

by double-clicking the object and even rotated (numerically) through the box opened by clicking 'Rotate' in the Draw pull-down menu. After creating and positioning the objects, you can make a CSG model by editing the set formula appropriately in the set formula field of the second line below the top menu-bar to take the union (by default), the intersection, and the set difference of the objects to form the shape of the domain Ω (Fig. 9.11). If you want to see the overall shape of the domain you created, click 'Boundary_mode' in the Boundary pull-down menu.

2. In Boundary mode, you can remove the subdomain borders that are induced by the intersections of the solid objects, but are not between different materials and also specify the boundary condition for each boundary segment. First, click the $\partial\Omega$ button in the tool-bar (Fig. 9.11) or 'Boundary_mode(^B)' in the Boundary pull-down menu (Fig. 9.12c), which will make the boundary segments appear with red/blue/green colors (indicating Dirichlet(default)/Neumann/mixed type of boundary condition) and arrows toward its end (for the case where the boundary condition is parameterized along the boundary). When you want to remove all the subdomain borders, click 'Remove_All_Subdomain_Borders' in the Boundary pull-down menu. You can set the parameters h, r or g, q in Eq. (9.5.2) to a constant or a function of x and y specifying the boundary condition, through the box window opened by double-clicking each boundary segment. In case you want to specify/change the boundary condition for multiple segments at a time, you had better use shift-click the segments to select all of them (which will be colored black) and click again on one of them to get the boundary condition dialog box.

3. In PDE mode, you can specify the type of PDE (Elliptic/Parabolic/Hyperbolic/Eigenmode) and its parameters. In order to do so, open the PDE specification dialog box by clicking the PDE button in the tool-bar or 'PDE_Specification' in the PDE pull-down menu (Fig. 9.12d), check the type of PDE, and set its parameters in Eq. (9.5.1)/(9.5.5)/(9.5.6)/(9.5.7).

4. In Mesh mode, you can create the triangular mesh for the domain drawn in Draw mode by just clicking the Δ button in the tool-bar or 'Initialize_Mesh(^I)' in the Mesh pull-down menu (Fig. 9.12e). To improve the accuracy of the solution, you can refine successively the mesh by clicking the ⟁ button in the tool-bar or 'Refine_Mesh(^M)' in the Mesh pull-down menu. You can jiggle the mesh by clicking 'Jiggle_Mesh' in expectation of better accuracy. You can also undo any refinement by clicking 'Undo_Mesh_Change' in the Mesh pull-down menu.

5. In Solve mode, you can solve the PDE and plot the result by just clicking the $=$ button in the tool-bar or 'Solve_PDE(^E)' in the Solve pull-down (Fig. 9.12f). But, in the case of parabolic or hyperbolic PDE, you must click 'Parameters' in the Solve pull-down menu (Fig. 9.12f) to set up the initial conditions and the time range before solving the PDE.

6. In Plot mode, you can change the plot option in the Plot selection dialog box opened by clicking the button in the tool-bar or 'Parameters' in the Plot pull-down menu (Fig. 9.12g). In the Plot selection dialog box (Fig. 9.12h), you can set the plot type to, say, Color/Height(3-D) and set the plot style to, say, interpolated shading and continuous (interpolated) height. If you want the mesh to be shown in the solution graph, check the box of Show_mesh. In case you want to plot the graph of a known function, change the option(s) of the Property into 'user_entry', type in the MATLAB expression describing the function and click the Plot button. You can save the plot parameters as the current default by clicking the Done button. You can also change the color map in the second line from the bottom of the dialog box.

(cf) We can extract the parameters involved in the domain geometry by clicking 'Export..' in the Draw pull-down menu, the parameters specifying the boundary by clicking 'Export..' in the Boundary pull-down menu, the parameters specifying the PDE by clicking 'Export..' in the PDE pull-down menu, the parameters specifying the mesh by clicking 'Export..' in the Mesh pull-down menu, the parameters related to the solution by clicking 'Export..' in the Solve pull-down menu, and the parameters related to the graph by clicking 'Export..' in the Plot pull-down menu. Whenever you want to save what you have worked in PDEtool, you may select File/Save in the top menu-bar.

(cf) Visit the website "http://www.mathworks.com/access/helpdesk/help/helpdesk.html" for more details.

9.5.3 Examples of Using PDETOOL to Solve PDEs

In this section, we will make use of PDEtool to solve some PDE problems that were dealt with in the previous sections.

Example 9.7. Laplace's Equation: Steady-State Temperature Distribution Over a Plate. Consider the Laplace's equation (Example 9.1)

$$\nabla^2 u(x, y) = \frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = 0 \qquad \text{for } 0 \le x \le 4,\ 0 \le y \le 4 \tag{E9.7.1}$$

with the following boundary conditions.

$$u(0, y) = e^y - \cos y, \qquad u(4, y) = e^y \cos 4 - e^4 \cos y \tag{E9.7.2}$$

$$u(x, 0) = \cos x - e^x, \qquad u(x, 4) = e^4 \cos x - e^x \cos 4 \tag{E9.7.3}$$

The procedure for using PDEtool to solve this problem is as follows:

0. Type '`pdetool`' into the MATLAB command window to have the PDE toolbox window on the screen. Then, adjust the ranges of the x-axis and

the y-axis to [0 5] and [0 5], respectively, in the dialog box opened by clicking ‘Axes_Limits’ in the Options pull-down menu. You can also click ‘Axes_Equal’ in the Options pull-down menu to have the x axis and the y axis of equal scale so that a circle/square may not look like an ellipse/rectangle.

1. Click the □ button in the tool-bar and click-and-drag on the graphic region to create a rectangle of domain. Then, in the Object dialog box opened by double-clicking the rectangle, set the Left/Bottom/Width/Height to 0/0/4/4. In this case, you don’t have to construct a CSG model by editing the set formula, because the domain consists of a single object: a rectangle.
2. Click the ∂Ω button in the tool-bar and double-click each boundary segment to specify the boundary condition as Eqs. (E9.7.2,3) in the boundary condition dialog box (see Fig. 9.13a).
3. Open the PDE specification dialog box by clicking the PDE button in the tool-bar, check the box on the left of Elliptic as the type of PDE, and set its parameters in Eq. (E9.7.1) as depicted in Fig. 9.13b.
4. Click the Δ button in the tool-bar to divide the domain into a number of triangular subdomains to get the triangular mesh as depicted in Fig. 9.13c. You can click the Δ button in the tool-bar to refine the mesh successively for better accuracy.
5. Click the = button in the tool-bar to plot the solution in the form of two-dimensional graph with the value of $u(x, y)$ shown in color.
6. If you want to plot the solution in the form of a three-dimensional graph with the value of $u(x, y)$ shown in height as well as color, check the box before Height on the far-left side of the Plot selection dialog box opened by clicking the button in the tool-bar. If you want the mesh shown in the solution plot as Fig. 9.13d, check the box before Show_mesh on the far-left and low side and click the Plot button at the bottom of the Plot selection dialog box (Fig. 9.12h). You can compare the result with that of Example 9.3 depicted in Fig. 9.4.
7. If you have the true analytical solution

$$u(x, y) = e^y \cos x - e^x \cos y \tag{E9.7.4}$$

and you want to plot the difference between the PDEtool (FEM) solution and the true analytical solution, change the entry ‘u’ into ‘user entry’ in the Color/Contour row and the Height row of the Property column and write ‘`u-(exp(y).*cos(x)-exp(x).*cos(y))`’ into the corresponding fields in the User_entry column of the Plot selection dialog box opened by clicking the button in the tool-bar and click the Plot button at the bottom of the dialog box.

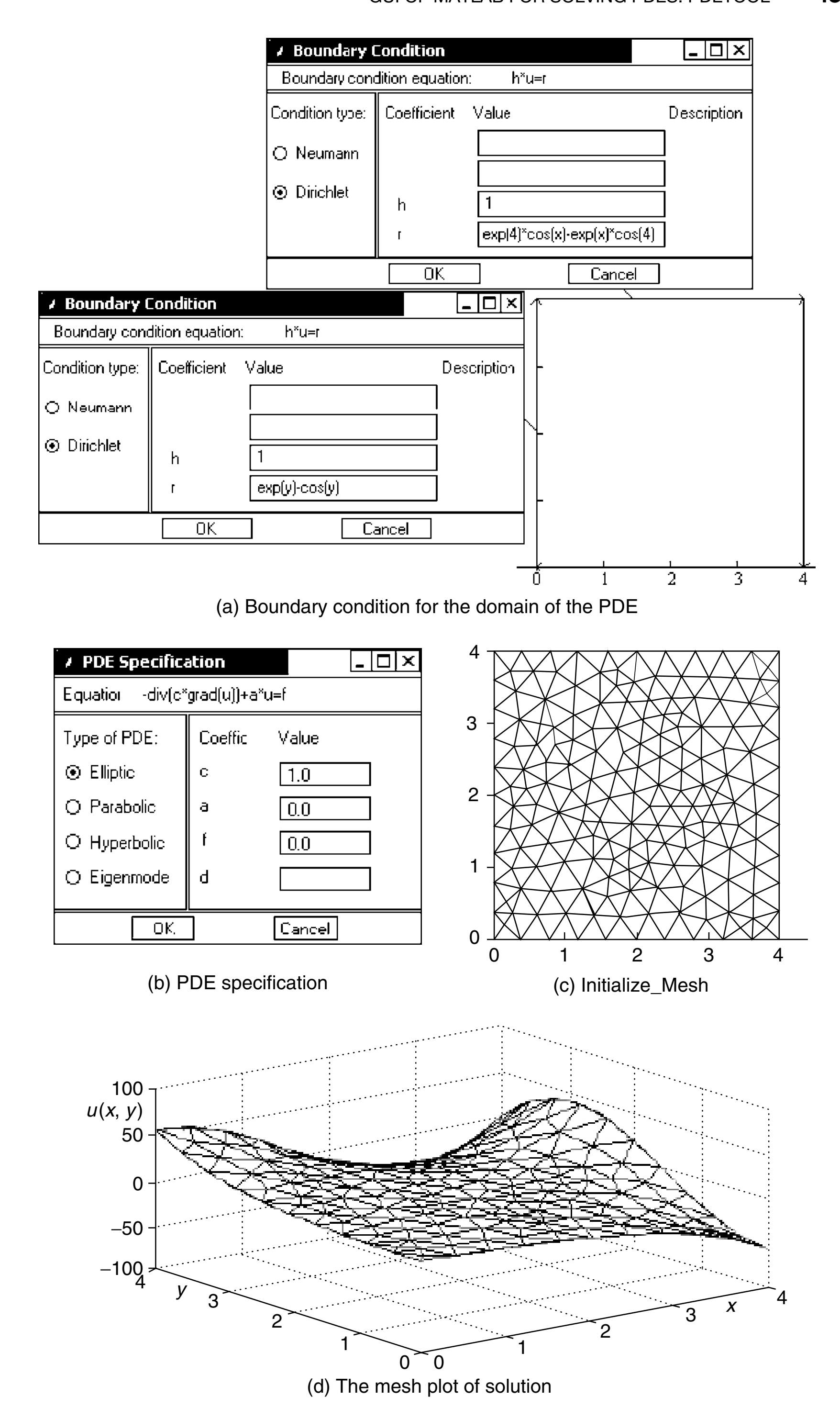


Figure 9.13 Procedure and results of using PDEtool for Example 9.1/9.7.

Example 9.8. A Parabolic PDE: Two-Dimensional Temperature Diffusion Over a Plate. Consider a two-dimensional parabolic PDE

$$10^{-4}\left(\frac{\partial^2 u(x,y,t)}{\partial x^2}+\frac{\partial^2 u(x,y,t)}{\partial y^2}\right)=\frac{\partial u(x,y,t)}{\partial t}$$
$$\text{for}\quad 0\le x\le 4,\quad 0\le y\le 4\quad \&\quad 0\le t\le 5000 \tag{E9.8.1}$$

with the initial conditions and boundary conditions

$$u(x,y,0)=0 \qquad \text{for } t=0 \tag{E9.8.2a}$$
$$u(x,y,t)=e^y\cos x-e^x\cos y\quad \text{for } x=0,\ x=4,\ y=0,\ y=4 \tag{E9.8.2b}$$

The procedure for using the PDEtool to solve this problem is as follows.

0–2. Do exactly the same things as steps 0–2 for the case of an elliptic PDE in Example 9.7.

3. Open the PDE specification dialog box by clicking the PDE button, check the box on the left of 'Parabolic' as the type of PDE and set its parameters in Eq. (E9.8.1) as depicted in Fig. 9.14a.
4. Exactly as in step 4 (for the case of elliptic PDE) in Example 9.7, click the Δ button to get the triangular mesh. You can click the button to refine the mesh successively for better accuracy.
5. Unlike the case of an elliptic PDE, you must click 'Parameters' in the Solve pull-down menu (Fig. 9.12f) to set the time range, say, as 0:100:5000 and the initial conditions as Eq. (E9.8.2a) before clicking the = button to solve the PDE. (See Fig. 9.14b.)
6. As in step 6 of Example 9.7, you can check the box before Height in the Plot selection dialog box opened by clicking the button, check the box before Show_mesh, and click the Plot button. If you want to plot the solution graph at a time other than the final time, select the time for plot from

   $$\{0, 100, 200, \ldots, 500\}$$

   in the far-right field of the Plot selection dialog box and click the Plot button again. If you want to see a movie-like dynamic picture of the solution graph, check the box before Animation, click Options right after Animation, fill in the fields of animation rate in fps (i.e., the number of frames per second and the number of repeats in the Animation Options dialog box), click the OK button, and then click the Plot button in the Plot selection dialog box.

(cf) If the dynamic picture is too oblong, you can scale up/down the solution by changing the Property of the Height row from 'u' into 'user entry' and filling in the corresponding field of User_entry with, say, 'u/25' in the Plot selection dialog box.

PDE Specification

Equation: d*u'-div(c*grad(u))+a*u=f

Type of PDE:	Coefficient	Value
○ Elliptic	c	1e-4
◉ Parabolic	a	0.0
○ Hyperbolic	f	0.0
○ Eigenmode	d	1.0

OK Cancel

(a) PDE specification dialog window

Solve Parameters

Time: 0:100:5000

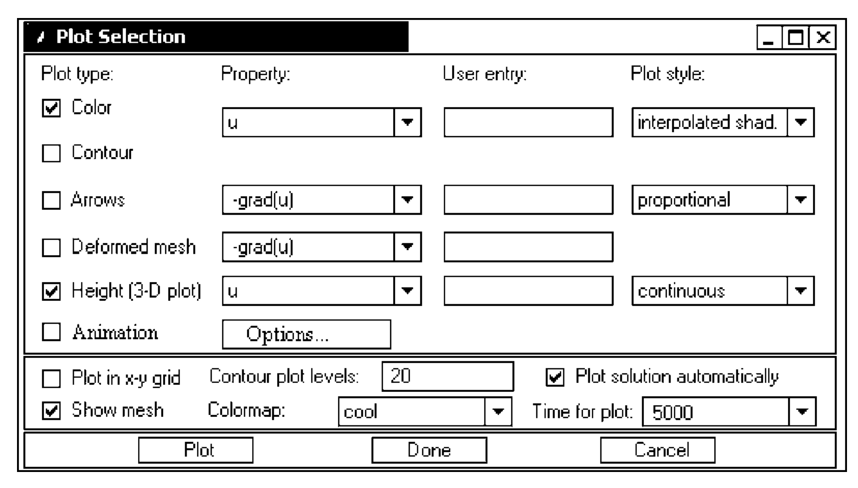
u(t0): 0.0

Relative tolerance: 0.01

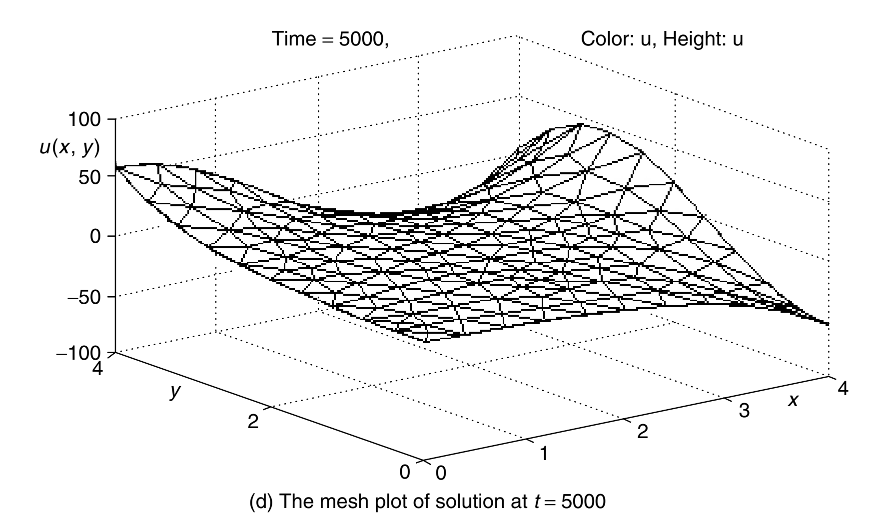
Absolute tolerance: 0.001

OK Cancel

(b) Solve parameters dialog window



(c) The box window popped out by clicking Parameters in the Plot pull-down menu



(d) The mesh plot of solution at $t = 5000$

Figure 9.14 Procedure and results of using PDEtool for Example 9.3/9.8.

According to your selection, you will see a movie-like dynamic picture or the (final) solution graph like Fig. 9.14d, which is the steady-state solution for Eq. (E9.8.1) with $\partial u(x, y, t)/\partial t = 0$, virtually the same as the elliptic PDE (E9.7.1) whose solution is depicted in Fig. 9.13d.

Before closing this example, let us have an experience of exporting the values of some parameters. For example, we extract the mesh data {p, e, t} by clicking 'Export_Mesh' in the Mesh pull-down menu and then clicking the OK button in the Export dialog box. Among the mesh data, the matrix p contains the x and y coordinates in the first and second rows, respectively. We also extract the solution u by clicking 'Export_Solution' in the Solve pull-down menu and then clicking the OK button in the Export dialog box. Now, we can estimate how far the graphical/numerical solution deviates from the true steady-state solution $u(x, y) = e^y \cos x - e^x \cos y$ by typing the following statements into the MATLAB command window.

```
>>x = p(1.:)'; y = p(2.: )'; %x.y coordinates of nodes in column vector
>>err = exp(y).*cos(x) - exp(x).*cos(y) - u(:.end); %deviation from true sol
>>err_max = max(abs(err)) %maximum absolute error
```

Note that the dimension of the solution matrix u is 177×51 and the solution at the final stage is stored in its last column `u(:,end)`, where 177 is the number of nodes in the triangular mesh and $51 = 5000/100 + 1$ is the number of frames or time stages.

Example 9.9. A Hyperbolic PDE: Two-Dimensional Wave (Vibration) Over a Square Membrane. Consider a two-dimensional hyperbolic PDE

$$\frac{1}{4}\left(\frac{\partial^2 u(x, y, t)}{\partial x^2} + \frac{\partial^2 u(x, y, t)}{\partial y^2}\right) = \frac{\partial u^2(x, y, t)}{\partial t^2}$$
$$\text{for} \quad 0 \le x \le 2,\ 0 \le y \le 2,\ \text{and}\ 0 \le t \le 2 \qquad \text{(E9.9.1)}$$

with the zero boundary conditions and the initial conditions

$$u(0, y, t) = 0, \quad u(2, y, t) = 0, \quad u(x, 0, t) = 0, \quad u(x, 2, t) = 0 \qquad \text{(E9.9.2)}$$

$$u(x, y, 0) = 0.1 \sin(\pi x) \sin(\pi y/2), \quad \partial u/\partial t(x, y, 0) = 0 \quad \text{for } t = 0 \qquad \text{(E9.9.3)}$$

The procedure for using the PDEtool to solve this problem is as follows:

0–2. Do the same things as steps 0–2 for the case of elliptic PDE in Example 9.7, except for the following.
- Set the ranges of the x axis-and the y-axis to [0 3] and [0 3].
- Set the Left/Bottom/Width/Height to 0/0/2/2 in the Object dialog box opened by double-clicking the rectangle.
- Set the boundary condition to zero as specified by Eqs. (E9.9.2) in the boundary condition dialog box opened by clicking the $\partial\Omega$ button in the tool-bar, shift-clicking the four boundary segments and double-clicking one of the boundary segments.

PDE Specification

Equation: d*u''-div(c*grad(u))+a*u=f

Type of PDE:	Coefficient	Value
○ Elliptic	c	1/4
○ Parabolic	a	0.0
◉ Hyperbolic	f	0.0
○ Eigenmode	d	1.0

OK Cancel

(a) PDE specification dialog window

Solve Parameters

Time: 0:0.1:2

u(t0): 0.1*sin(pi*x).*sin(pi*y/2)

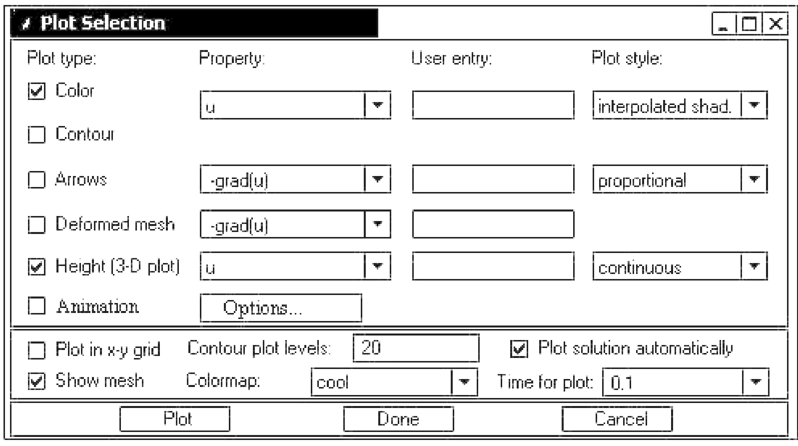
u'(t0): 0.0

Relative tolerance: 0.01

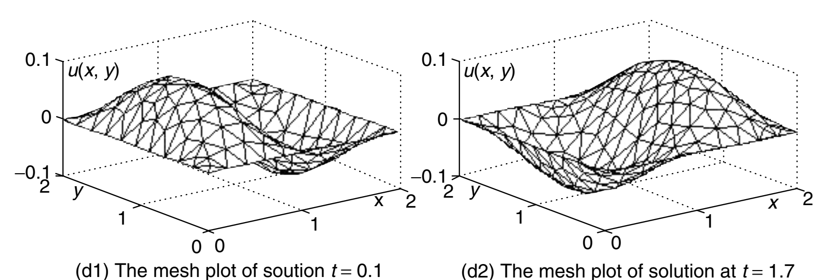
Absolute tolerance: 0.001

OK Cancel

(b) Solve parameters dialog window



(c) The box window popped out by clicking Parameters in the Plot pull-down menu



(d1) The mesh plot of soution $t = 0.1$ (d2) The mesh plot of solution at $t = 1.7$

Figure 9.15 Procedure and results of using PDEtool for Example 9.5/9.9.

3. Open the PDE specification dialog box by clicking the PDE button, check the box on the left of ‘Hyperbolic’ as the type of PDE, and set its parameters in Eq. (E9.9.1) as depicted in Fig. 9.15a.
4. Do the same thing as step 4 for the case of elliptic PDE in Example 9.8.
5. Similarly to the case of a parabolic PDE, you must click ‘Parameters’ in the Solve pull-down menu (Fig. 9.12f) to set the time range, say, as

0:0.1:2 and the initial conditions as Eq. (E9.9.3) before clicking the = button to solve the PDE. (See Fig. 9.15b.)

6. Do almost the same thing as step 6 for the case of parabolic PDE in Example 9.8.

Finally, you could see the solution graphs like Figs. 9.15(d1)&(d2), that are similar to Figs. 9.6(a)&(c).

Example 9.10. Laplace's Equation: Electric Potential Over a Plate with Point Charge. Consider the Laplace's equation (dealt with in Example 9.6)

$$\nabla^2 u(x,y) = \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = f(x,y)$$
$$\text{for} \quad -1 \le x \le +1, -1 \le y \le +1 \tag{E9.10.1}$$

where

$$f(x,y) = \begin{cases} -1 & \text{for } (x,y) = (0.5, 0.5) \\ +1 & \text{for } (x,y) = (-0.5, -0.5) \\ 0 & \text{elsewhere} \end{cases} \tag{E9.10.2}$$

and the boundary condition is $u(x,y) = 0$ for all boundaries of the rectangular domain.

The procedure for using the PDEtool to solve this problem is as follows.

0–2. Do the same thing as step 0–2 for the case of elliptic PDE in Example 9.7, except for the following.
   - Set the Left/Bottom/Width/Height to −1/−1/2/2 in the Object dialog box opened by double-clicking the rectangle.
   - Set the boundary condition to zero in the boundary condition dialog box opened by clicking the $\partial\Omega$ button in the tool-bar, shift-clicking the four boundary segments, and double-clicking one of the boundary segments.

3. Open the PDE specification dialog box by clicking the PDE button, check the box on the left of 'Elliptic' as the type of PDE, and set its parameters in Eq. (E9.10.1,2) as depicted in Fig. 9.16a.
4. Click the Δ button to initialize the triangular mesh.
5. Click the button to open the Plot selection dialog box, check the box before 'Height', and check the box before 'Show_mesh' in the dialog box.
6. Click the Plot button to get the solution graph as depicted in Fig. 9.16c.
7. Click 'Parameters' in the Solve pull-down menu to open the 'Solve Parameters' dialog box depicted in Fig. 9.16b, check the box on the left of 'Adaptive mode', and click the OK button in order to activate the adaptive mesh mode.
8. Click the = button to get a solution graph with the adaptive mesh.

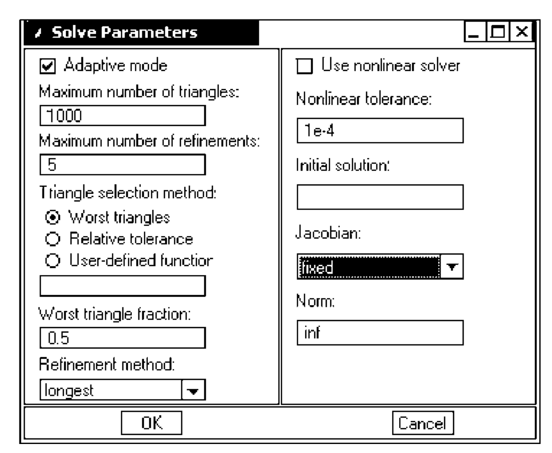
PDE Specification

Equation: -div(c*grad(u))+a*u=f

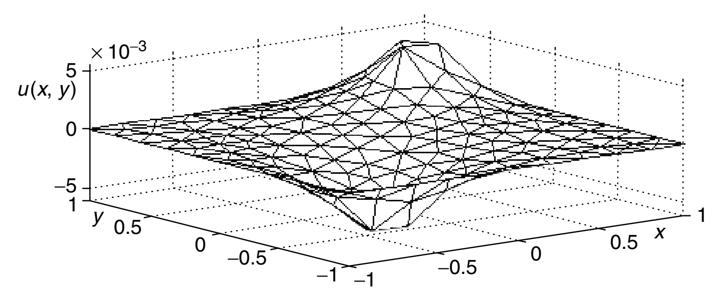
Type of PDE:	Coefficient	Value
⦿ Elliptic	c	1
○ Parabolic	a	0
○ Hyperbolic	f	(-(((x+0.5).^2+(y+0.5).^2)<0.01)+(((x-0.5).^2+(y-0.5).^2)<0.01))
○ Eigenmode	d	

OK Cancel

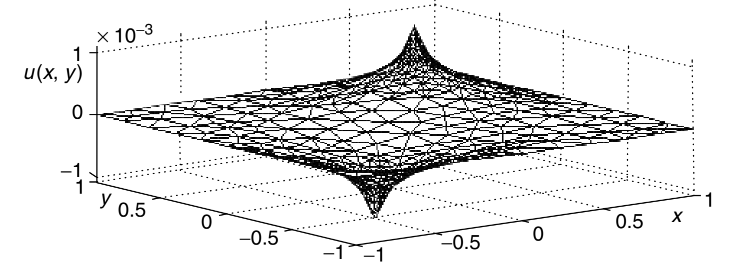
(a) PDE specification dialog window



(b) Solve parameters dialog window



(c) The mesh plot of the solution with Initialize mesh



(d) The mesh plot of the solution obtained by using Adaptive mesh one time

Figure 9.16 Procedure and results of using PDEtool for Example 9.6/9.10.

9. Noting that the solution is not the right one for the point charge distribution given by (E9.10.2), reopen the PDE specification dialog box by clicking the PDE button and rewrite f as below.

```
f  (-(((x + 0.5).^2 + (y + 0.5).^2 < 0.00064) + (((x - 0.5).^2 + (y - 0.5).^2) < 0.00064))
```

10. Noting that the mesh has already been refined in the adaptive way to yield smaller meshes in the region where the slope of the solution is steeper, click 'Parameters' in the Solve pull-down menu to open the 'Solve Parameters' dialog box, uncheck the box on the left of 'Adaptive mode', and click the OK button in the dialog box in order to inactivate the adaptive mesh mode.
11. Click the = button to get the solution graph as depicted in Fig. 9.16d.
12. You can click 'Refine_Mesh(^M)' in the Mesh pull-down menu and click the = button to get a more refined solution graph (with higher resolution) as many times as you want.

PROBLEMS

9.1 Elliptic PDEs: Poisson Equations

Use the routine "`poisson()`" (in Section 9.1) to solve the following PDEs and plot the solutions by using the MATLAB command "`mesh()`".

(a) $$\nabla^2 u(x, y) = \frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = x + y \quad \text{for } 0 \le x \le 1, 0 \le y \le 1 \tag{P9.1.1}$$

with the boundary conditions

$$u(0, y) = y^2, \qquad u(1, y) = 1,$$
$$u(x, 0) = x^2, \qquad u(x, 1) = 1 \tag{P9.1.2}$$

Divide the solution region (domain) into $M_x \times M_y = 5 \times 10$ sections.

(b) $$\frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} - 12.5\pi^2 u(x, y) = -25\pi^2 \cos\left(\frac{5\pi}{2}x\right)\cos\left(\frac{5\pi}{2}y\right) \quad \text{for } 0 \le x, y \le 0.4 \tag{P9.1.3}$$

with the boundary conditions

$$u(0, y) = \cos\left(\frac{5\pi}{2}y\right), \qquad u(0.4, y) = -\cos\left(\frac{5\pi}{2}y\right) \tag{P9.1.4}$$

$$u(x, 0) = \cos\left(\frac{5\pi}{2}x\right), \qquad u(x, 0.4) = -\cos\left(\frac{5\pi}{2}x\right) \tag{P9.1.5}$$

Divide the solution region into $M_x \times M_y = 40 \times 40$ sections.

(c) $$\frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} + 4\pi(x^2 + y^2)u(x, y)$$
$$= 4\pi \cos(\pi(x^2 + y^2)) \qquad \text{for } 0 \le x \le 1, 0 \le y \le 1 \qquad \text{(P9.1.6)}$$

with the boundary conditions

$$u(0, y) = \sin(\pi y^2), \qquad u(1, y) = \sin(\pi(y^2 + 1)) \qquad \text{(P9.1.7)}$$
$$u(x, 0) = \sin(\pi x^2), \qquad u(x, 1) = \sin(\pi(x^2 + 1)) \qquad \text{(P9.1.8)}$$

Divide the solution region into $M_x \times M_y = 40 \times 40$ sections.

(d) $$\frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = 10e^{2x+y} \quad \text{for } 0 \le x \le 1, 0 \le y \le 2 \quad \text{(P9.1.9)}$$

with the boundary conditions

$$u(0, y) = 2e^y, \qquad u(1, y) = 2e^{2x+y},$$
$$u(x, 0) = 2e^{2x}, \qquad u(x, 2) = 2e^{2x+2} \qquad \text{(P9.1.10)}$$

Divide the solution region into $M_x \times M_y = 20 \times 40$ sections.

(e) $$\frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2} = 0 \qquad \text{for } 0 \le x \le 1, 0 \le y \le \pi/2 \quad \text{(P9.1.11)}$$

with the boundary conditions

$$u(0, y) = 4\cos(3y), \qquad u(1, y) = 4e^{-3}\cos(3y),$$
$$u(x, 0) = 4e^{-3x}, \qquad u(x, \pi/2) = 0 \qquad \text{(P9.1.12)}$$

Divide the solution region into $M_x \times M_y = 20 \times 20$ sections.

9.2 More General PDE Having Nonunity Coefficients

Consider the following PDE having nonunity coefficients.

$$A\frac{\partial^2 u(x, y)}{\partial x^2} + B\frac{\partial^2 u(x, y)}{\partial x \partial y} + C\frac{\partial^2 u(x, y)}{\partial y^2} + g(x, y)u(x, y) = f(x, y) \qquad \text{(P9.2.1)}$$

Modify the routine "`poisson()`" so that it can solve this kind of PDEs and declare it as

```
function [u,x,y] = poisson_abc(ABC,f,g,bx0,bxf,by0,...,Mx,My,tol,imax)
```

where the first input argument ABC is supposed to carry the vector containing three coefficients A, B, and C. Use the routine to solve the following PDEs and plot the solutions by using the MATLAB command “`mesh()`”.

(a) $$\frac{\partial^2 u(x,y)}{\partial x^2} + 2\frac{\partial^2 u(x,y)}{\partial y^2} = 10 \quad \text{for} \quad 0 \le x \le 1, 0 \le y \le 1 \tag{P9.2.2}$$

with the boundary conditions

$$\begin{aligned} u(0,y) &= y^2, \qquad u(1,y) = (y+2)^2, \\ u(x,0) &= 4x^2, \qquad u(1,y) = (2x+1)^2 \end{aligned} \tag{P9.2.3}$$

Divide the solution region (domain) into $M_x \times M_y = 20 \times 40$ sections.

(b) $$\frac{\partial^2 u(x,y)}{\partial x^2} + 3\frac{\partial^2 u(x,y)}{\partial x \partial y} + 2\frac{\partial^2 u(x,y)}{\partial y^2} = 0 \quad \text{for } 0 \le x \le 1, 0 \le y \le 1 \tag{P9.2.4}$$

with the boundary conditions

$$u(0,y) = e^y + \cos y, \quad u(1,y) = e^{y-1} + \cos(y-2) \tag{P9.2.5}$$

$$u(x,0) = e^{-x} + \cos(-2x), \quad u(x,1) = e^{1-x} + \cos(1-2x) \tag{P9.2.6}$$

Divide the solution region into $M_x \times M_y = 40 \times 40$ sections.

(c) $$\frac{\partial^2 u(x,y)}{\partial x^2} + 3\frac{\partial^2 u(x,y)}{\partial x \partial y} + 2\frac{\partial^2 u(x,y)}{\partial y^2} = x \sin y \quad \text{for} \quad 0 \le x \le 2, 0 \le y \le \pi \tag{P9.2.7}$$

with the boundary conditions

$$u(0,y) = (3/4)\cos y, \quad u(2,y) = -\sin(y) + (3/4)\cos y \tag{P9.2.8}$$

$$u(x,0) = 3/4, \qquad u(x,\pi) = -3/4 \tag{P9.2.9}$$

Divide the solution region into $M_x \times M_y = 20 \times 40$ sections.

(d) $$4\frac{\partial^2 u(x,y)}{\partial x^2} - 4\frac{\partial^2 u(x,y)}{\partial x \partial y} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0 \quad \text{for } 0 \le x \le 1, 0 \le y \le 1 \tag{P9.2.10}$$

with the boundary conditions

$$\begin{aligned} u(0,y) &= ye^{2y}, \quad u(1,y) = (1+y)e^{1+2y}, \\ u(x,0) &= xe^x, \quad u(x,1) = (x+1)e^{x+2} \end{aligned} \tag{P9.2.11}$$

Divide the solution region into $M_x \times M_y = 40 \times 40$ sections.

```
function [u,x,y] = poisson_Neuman(f,g,bx0,bxf,by0,byf,x0,xf,y0,yf,...)
.. .. .. .. .. .. .. ..
Neum = zeros(1,4); %Not Neumann, but Dirichlet condition by default
if length(x0) > 1, Neum(1) = x0(2); x0 = x0(1); end
if length(xf) > 1, Neum(2) = xf(2); xf = xf(1); end
if length(y0) > 1, Neum(3) = y0(2); y0 = y0(1); end
if length(yf) > 1, Neum(4) = yf(2); yf = yf(1); end
.. .. .. .. .. .. .. ..
dx_2 = dx*dx; dy_2 = dy*dy;  dxy2=2*(dx_2 + dy_2);
rx = dx_2/dxy2; ry = dy_2/dxy2; rxy = rx*dy_2; rx = rx;
dx2 = dx*2; dy2 = dy*2; rydx = ry*dx2; rxdy = rx*dy2;
u(1:My1,1:Mx1) = zeros(My1,Mx1);

sum_of_bv = 0; num = 0;
if Neum(1) == 0 %Dirichlet boundary condition
for m = 1:My1, u(m,1) = bx0(y(m)); end %side a
 else %Neumann boundary condition
  for m = 1:My1, duxa(m) = bx0(y(m)); end %du/dx(x0,y)
end
if Neum(2) == 0 %Dirichlet boundary condition
  .. .. .. .. .. .. .. .. .. .. ..
end
if Neum(3) == 0 %Dirichlet boundary condition
  n1 = 1; nM1 = Mx1;
  if Neum(1) == 0, u(1,1)=(u(1,1) + by0(x(1)))/2; n1 = 2; end
  if Neum(2) == 0, u(1,Mx1)=(u(1,Mx1) + by0(x(Mx1)))/2; nM1 = Mx; end
  for n = n1:nM1, u(1,n) = by0(x(n)); end %side c
 else %Neumann boundary condition
  for n = 1:Mx1, duyc(n) = by0(x(n)); end %du/dy(x,y0)
end
if Neum(4) == 0 %Dirichlet boundary condition
  .. .. .. .. .. .. .. .. .. .. ..
end
for itr = 1:imax
  if Neum(1) %Neumann boundary condition
    for i = 2:My
       u(i,1) = 2*ry*u(i,2) + rx*(u(i + 1,1) + u(i-1,1)) ...
                   + rxy*(G(i,1)*u(i,1) - F(i,1)) - rydx*duxa(i); %(9.1.9)
    end
    if Neum(3), u(1,1) = 2*(ry*u(1,2) +rx*u(2,1)) ...
     + rxy*(G(1,1)*u(1,1) - F(1,1)) - rydx*duxa(1) - rxdy*duyc(1);%(9.1.11)
    end
    if Neum(4), u(My1,1) = 2*(ry*u(My1,2) +rx*u(My,1)) ...
     + rxy*(G(My1,1)*u(My1,1)- F(My1,1))+rxdy*duyd(1)- rydx*duxa(My1);
    end
  end
  if Neum(2) %Neumann boundary condition
  .. .. .. .. .. .. .. .. .. .. .. .. ..
  end
  if Neum(3) %Neumann boundary condition
    for j = 2:Mx
       u(1,j) = 2*rx*u(2,j)+ry*(u(1,j+1) + u(1,j-1)) ...
                  +rxy*(G(1,j)*u(1,j) - F(1,j)) - rxdy*duyc(j); %(9.1.10)
    end
  end
  if Neum(4) %Neumann boundary condition
   .. .. .. .. .. .. .. .. .. .. .. .. ..
  end
  .. .. .. .. .. .. .. .. .. .. .. .. ..
end
```

9.3 Elliptic PDEs with Neumann Boundary Condition

Consider the PDE (E9.1.1) (dealt with in Example 9.1)

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0 \qquad \text{for } 0 \le x \le 4, 0 \le y \le 4 \tag{P9.3.1}$$

with different boundary conditions of Neumann type, which was discussed in Section 9.1. Modify the routine "`poisson()`" so that it can deal with the Neumann boundary condition and declare it as

```
function [u,x,y] = poisson_Neuman(f,g,bx0,bxf,by0,byf,x0,xf,y0,yf,...)
```

where the third/fourth/fifth/sixth input arguments are supposed to carry the functions of

$$u(x_0, y)/u(x_f, y)/u(x, y_0)/u(x, y_f)$$

or

$$\partial u(x,y)/\partial x|_{x=x_0}/\partial u(x,y)/\partial x|_{x=x_f}/\partial u(x,y)/\partial y|_{y=y_0}/\partial u(x,y)/\partial y|_{y=y_f}$$

and the seventh/eighth/ninth/tenth input arguments are to carry $x_0/x_f/y_0/y_f$ or [x_0 1]/[x_f 1]/[y_0]/[y_f 1] depending on whether each boundary condition is of Dirichlet or Neumann type. Use it to solve the PDE with the following boundary conditions and plot the solutions by using the MATLAB command "`mesh()`". Divide the solution region (domain) into $M_x \times M_y = 20 \times 20$ sections.

(cf) You may refer to the related part of the program in the previous page.

(a) $\partial u(x,y)/\partial x|_{x=0} = -\cos y, \quad u(4,y) = e^y \cos 4 - e^4 \cos y$ (P9.3.2)

$\partial u(x,y)/\partial y|_{y=0} = \cos x, \quad u(x,4) = e^4 \cos x - e^x \cos 4$ (P9.3.3)

(b) $u(0,y) = e^y - \cos y, \ \partial u(x,y)/\partial x|_{x=4} = -e^y \sin 4 - e^4 \cos y$ (P9.3.4)

$u(x,0) = \cos x - e^x, \ \partial u(x,y)/\partial y|_{y=4} = e^4 \cos x + e^x \sin 4$ (P9.3.5)

(c) $\partial u(x,y)/\partial x|_{x=0} = -\cos y, \quad u(4,y) = e^y \cos 4 - e^4 \cos y$ (P9.3.6)

$u(x,0) = \cos x - e^x, \quad \partial u(x,y)/\partial y\big|_{y=4} = e^4 \cos x + e^x \sin 4$ (P9.3.7)

(d) $u(0,y) = e^y - \cos y, \ \partial u(x,y)/\partial x\big|_{x=4} = -e^y \sin 4 - e^4 \cos y$ (P9.3.8)

$\partial u(x,y)/\partial y\big|_{y=0} = \cos x, \qquad u(x,4) = e^4 \cos x - e^x \cos 4$ (P9.3.9)

(e) $\partial u(x, y)/\partial x\big|_{x=0} = -\cos y, \quad \partial u(x, y)/\partial x\big|_{x=4} = -e^y \sin 4 - e^4 \cos y$ (P9.3.10)

$\partial u(x, y)/\partial y\big|_{y=0} = \cos x, \quad u(x, 4) = e^4 \cos x - e^x \cos 4$ (P9.3.11)

(f) $\partial u(x, y)/\partial x\big|_{x=0} = -\cos y, \quad u(4, y) = e^y \cos 4 - e^4 \cos y$ (P9.3.12)

$\partial u(x, y)/\partial y\big|_{y=0} = \cos x, \quad \partial u(x, y)/\partial y\big|_{y=4} = e^4 \cos x + e^x \sin 4$ (P9.3.13)

(g) $u(0, y) = e^y - \cos y, \; \partial u(x, y)/\partial x\big|_{x=4} = -e^y \sin 4 - e^4 \cos y$ (P9.3.14)

$\partial u(x, y)/\partial y\big|_{y=0} = \cos x, \quad \partial u(x, y)/\partial y\big|_{y=4} = e^4 \cos x + e^x \sin 4$ (P9.3.15)

(h) $\partial u(x, y)/\partial x\big|_{x=0} = -\cos y, \quad \partial u(x, y)/\partial x\big|_{x=4} = -e^y \sin 4 - e^4 \cos y$ (P9.3.16)

$\partial u(x, y)/\partial y\big|_{y=0} = \cos x, \quad \partial u(x, y)/\partial y\big|_{y=4} = e^4 \cos x + e^x \sin 4$ (P9.3.17)

9.4 Parabolic PDEs: Heat Equations

Modify the program "`solve_heat.m`" (in Section 9.2.3) so that it can solve the following PDEs by using the explicit forward Euler method, the implicit backward Euler method, and the Crank–Nicholson method.

(a) $$\frac{\partial^2 u(x, t)}{\partial x^2} = \frac{\partial u(x, t)}{\partial t} \quad \text{for } 0 \le x \le 1, 0 \le t \le 0.1 \tag{P9.4.1}$$

with the initial/boundary conditions

$$u(x, 0) = x^4, \qquad u(0, t) = 0, \qquad u(1, t) = 1 \tag{P9.4.2}$$

(i) With the solution region divided into $M \times N = 10 \times 20$ sections, does the explicit forward Euler method converge? What is the value of $r = A\Delta t/(\Delta x)^2$?

(ii) If you increase M and N to make $M \times N = 20 \times 40$ for better accuracy, does the explicit forward Euler method still converge? What is the value of $r = A\Delta t/(\Delta x)^2$?

(iii) What is the number N of subintervals along the t axis that we should choose in order to keep the same value of r for $M = 20$? With that value of r, does the explicit forward Euler method converge?

(b) $10^{-5}\dfrac{\partial^2 u(x,t)}{\partial x^2} = \dfrac{\partial u(x,t)}{\partial t}$ for $0 \le x \le 1, 0 \le t \le 6000$ (P9.4.3)

with the initial/boundary conditions

$$u(x,0) = 2x + \sin(2\pi x), \quad u(0,t) = 0, \quad u(1,t) = 2 \qquad \text{(P9.4.4)}$$

(i) With the solution region divided into $M \times N = 20 \times 40$ sections, does the explicit forward Euler method converge? What is the value of $r = A\Delta t/(\Delta x)^2$? Does the numerical stability condition (9.2.6) seem to be so demanding?

(ii) If you increase M and N to make $M \times N = 40 \times 160$ for better accuracy, does the explicit forward Euler method still converge? What is the value of $r = A\Delta t/(\Delta x)^2$? Does the numerical stability condition (9.2.6) seem to be so demanding?

(iii) With the solution region divided into $M \times N = 40 \times 200$ sections, does the explicit forward Euler method converge? What is the value of $r = A\Delta t/(\Delta x)^2$?

(c) $2\dfrac{\partial^2 u(x,t)}{\partial x^2} = \dfrac{\partial u(x,t)}{\partial t}$ for $0 \le x \le \pi, 0 \le t \le 0.2$ (P9.4.5)

with the initial/boundary conditions

$$u(x,0) = \sin(2x), \qquad u(0,t) = 0, \qquad u(\pi,t) = 0 \qquad \text{(P9.4.6)}$$

(i) By substituting

$$u(x,t) = \sin(2x)e^{-8t} \qquad \text{(P9.4.7)}$$

into the above equation (P9.4.5), verify that this is a solution to the PDE.

(ii) With the solution region divided into $M \times N = 40 \times 100$ sections, does the explicit forward Euler method converge? What is the value of $r = A\Delta t/(\Delta x)^2$?

(iii) If you increase N (the number of subintervals along the t-axis) to 125 for improving the numerical stability, does the explicit forward Euler method converge? What is the value of $r = A\Delta t/(\Delta x)^2$? Use the MATLAB statements in the following box to find the maximum absolute errors of the numerical solutions obtained by the three methods. Which method yields the smallest error?

```
uo = inline('sin(2*x)*exp(-8*t)','x','t'); %true analytical solution
Uo = uo(x,t);
err = max(max(abs(u1 - Uo)))
```

(iv) If you increase N to 200, what is the value of $r = A\Delta t/(\Delta x)^2$? Find the maximum absolute errors of the numerical solutions obtained by the three methods as in (iii). Which method yields the smallest error?

(d) $$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \quad \text{for } 0 \le x \le 1, 0 \le t \le 0.1 \tag{P9.4.8}$$

with the initial/boundary conditions

$$u(x,0) = \sin(\pi x) + \sin(3\pi x), \quad u(0,t) = 0, \quad u(1,t) = 0 \tag{P9.4.9}$$

(i) By substituting

$$u(x,t) = \sin(\pi x)e^{-\pi^2 t} + \sin(3\pi x)e^{-(3\pi)^2 t} \tag{P9.4.10}$$

into Eq. (P9.4.5), verify that this is a solution to the PDE.

(ii) With the solution region divided into $M \times N = 25 \times 80$ sections, does the explicit forward Euler method converge? What is the value of $r = A\Delta t/(\Delta x)^2$?

(iii) If you increase N (the number of subintervals along the t axis) to 100 for improving the numerical stability, does the explicit forward Euler method converge? What is the value of $r = A\Delta t/(\Delta x)^2$? Find the maximum absolute errors of the numerical solutions obtained by the three methods as in (c)(iii).

(iv) If you increase N to 200, what is the value of $r = A\Delta t/(\Delta x)^2$? Find the maximum absolute errors of the numerical solutions obtained by the three methods as in (c)(iii). Which one gained the accuracy the most of the three methods through increasing N?

9.5 Parabolic PDEs with Neumann Boundary Conditions

Let us modify the routines "`heat_exp()`", "`heat_imp()`", and "`heat_cn()`" (in Section 9.2) so that they can accommodate the heat equation (9.2.1) with Neumann boundary conditions

$$\partial u(x,t)/\partial x\big|_{x=x_0} = b_{x_0}(t), \quad \partial u(x,t)/\partial x\big|_{x=x_f} = b_{x_f}(t) \tag{P9.5.1}$$

(a) Consider the explicit forward Euler algorithm described by Eq. (9.2.3)

$$u_i^{k+1} = r(u_{i+1}^k + u_{i-1}^k) + (1-2r)u_i^k$$
$$\text{for } i = 1, 2, \ldots, M-1 \text{ with } r = A\frac{\Delta t}{\Delta x^2} \tag{P9.5.2}$$

In the case of Dirichlet boundary condition, we don't need to get u_0^{k+1} and u_M^{k+1}, because they are already given. But, in the case of the Neumann boundary condition, we must get them by using this equation for $i = 0$ and M as

$$u_0^{k+1} = r(u_1^k + u_{-1}^k) + (1 - 2r)u_0^k \tag{P9.5.3a}$$

$$u_M^{k+1} = r(u_{M+1}^k + u_{M-1}^k) + (1 - 2r)u_M^k \tag{P9.5.3b}$$

and the boundary conditions approximated as

$$\frac{u_1^k - u_{-1}^k}{2\Delta x} = b_0'(k), \qquad u_{-1}^k = u_1^k - 2b_0'(k)\Delta x \tag{P9.5.4a}$$

$$\frac{u_{M+1}^k - u_{M-1}^k}{2\Delta x} = b_M'(k), \qquad u_{M+1}^k = u_{M-1}^k + 2b_M'(k)\Delta x \tag{P9.5.4b}$$

Substituting Eqs. (P9.5.4a,b) into Eq. (P9.5.3) yields

$$u_0^{k+1} = 2r(u_1^k - b_0'(k)\Delta x) + (1 - 2r)u_0^k \tag{P9.5.5a}$$

$$u_M^{k+1} = 2r(u_{M-1}^k + b_M'(k)\Delta x) + (1 - 2r)u_M^k \tag{P9.5.5b}$$

Modify the routine "`heat_exp()`" so that it can use this scheme to deal with the Neumann boundary conditions for solving the heat equation and declare it as

```
function [u,x,t] = heat_exp_Neuman(a,xfn,T,it0,bx0,bxf,M,N)
```

where the second input argument `xfn` and the fifth and sixth input arguments `bx0,bxf` are supposed to carry `[xf 0 1]` and $b_{x_0}(t)$, $b'_{x_f}(t)$, respectively, if the boundary condition at x_0/x_f is of Dirichlet/Neumann type and they are also supposed to carry `[xf 1 1]` and $b'_{x_0}(t)$, $b'_{x_f}(t)$, respectively, if both of the boundary conditions at x_0/x_f are of Neumann type.

(b) Consider the implicit backward Euler algorithm described by Eq. (9.2.13), which deals with the Neumann boundary condition at the one end for solving the heat equation (9.2.1). With reference to Eq. (9.2.13), modify the routine "`heat_imp()`" so that it can solve the heat equation with the Neumann boundary conditions at two end points x_0 and x_f and declare it as

```
function [u,x,t] = heat_imp_Neuman(a,xfn,T,it0,bx0,bxf,M,N)
```

(c) Consider the Crank–Nicholson algorithm described by Eq. (9.2.17), which deals with the Neumann boundary condition at the one end for solving the heat equation (9.2.1). With reference to Eq. (9.2.17), modify the routine "`heat_cn()`" so that it can solve the heat equation with the Neumann boundary conditions at two end points x_0 and x_f and declare it as

```
function [u,x,t] = heat_cn_Neuman(a,xfn,T,it0,bx0,bxf,M,N)
```

(d) Solve the following heat equation with three different boundary conditions by using the three modified routines in (a), (b), (c) with `M = 20,N = 100` and find the maximum absolute errors of the three solutions as in Problem 9.4(c)(iii).

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \quad \text{for } 0 \le x \le 1, 0 \le t \le 0.1 \tag{P9.5.6}$$

with the initial/boundary conditions

(i) $u(x,0) = \sin(\pi x), \quad \partial u(x,t)/\partial x\big|_{x=0} = \pi\, e^{-\pi^2 t}, \quad u(x,t)\big|_{x=1} = 0$ (P9.5.7)

(ii) $u(x,0) = \sin(\pi x), \quad u(x,t)\big|_{x=0} = 0, \quad \partial u(x,t)/\partial x\big|_{x=1} = -\pi\, e^{-\pi^2 t}$ (P9.5.8)

(iii) $u(x,0) = \sin(\pi x), \quad \partial u(x,t)/\partial x\big|_{x=0} = \pi\, e^{-\pi^2 t},$

$$\partial u(x,t)/\partial x\big|_{x=1} = -\pi\ e^{-\pi^2 t} \tag{P9.5.9}$$

Note that the true analytical solution is

$$u(x,t) = \sin(\pi x) e^{-\pi^2 t} \tag{P9.5.10}$$

9.6 Hyperbolic PDEs: Wave Equations

Modify the program "`solve_wave.m`" (in Section 9.3) so that it can solve the following PDEs by using the explicit forward Euler method, the implicit backward Euler method, and the Crank–Nicholson method.

(a) $$4\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial^2 u(x,t)}{\partial t^2} \quad \text{for } 0 \le x \le 1, 0 \le t \le 1 \tag{P9.6.1}$$

with the initial/boundary conditions

$$\begin{aligned} u(x,0) &= 0, \qquad \partial u(x,t)/\partial t\big|_{t=0} = 5\sin(\pi x), \\ u(0,t) &= 0, \qquad u(1,t) = 0 \end{aligned} \tag{P9.6.2}$$

Note that the true analytical solution is

$$u(x,t) = \frac{2.5}{\pi}\sin(\pi x)\sin(2\pi t) \tag{P9.6.3}$$

(i) With the solution region divided into $M \times N = 20 \times 50$ sections, what is the value of $r = A(\Delta t)^2/(\Delta x)^2$? Use the MATLAB statements in Problem 9.4(c)(iii) to find the maximum absolute error of the solution obtained by using the routine "`wave()`".

(ii) With the solution region divided into $M \times N = 40 \times 100$ sections, what is the value of r? Find the maximum absolute error of the numerical solution.

(iii) If we increase M (the number of subintervals along the x axis) to 50 for better accuracy, what is the value of r? Find the maximum absolute error of the numerical solution and determine whether it has been improved.

(iv) If we increase the number M to 52, what is the value of r? Can we expect better accuracy in the light of the numerical stability condition (9.3.7)? Find the maximum absolute error of the numerical solution and determine whether it has been improved or not.

(v) What do you think the best value of r is?

(b) $$6.25\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial^2 u(x,t)}{\partial t^2} \quad \text{for } 0 \le x \le \pi,\ 0 \le t \le 0.4\pi \tag{P9.6.4}$$

with the initial/boundary conditions

$$u(x,0) = \sin(2x), \qquad \partial u(x,t)/\partial t\big|_{t=0} = 0,$$
$$u(0,t) = 0, \qquad u(1,t) = 0 \tag{P9.6.5}$$

Note that the true analytical solution is

$$u(x,t) = \sin(2x)\cos(5t) \tag{P9.6.6}$$

(i) With the solution region divided into $M \times N = 50 \times 50$ sections, what is the value of $r = A(\Delta t)^2/(\Delta x)^2$? Find the maximum absolute error of the solution obtained by using the routine "`wave()`".

(ii) With the solution region divided into $M \times N = 50 \times 49$ sections, what is the value of r? Find the maximum absolute error of the numerical solution.

(iii) If we increase N (the number of subintervals along the t axis) to 51 for better accuracy, what is the value of r? Find the maximum absolute error of the numerical solution.

(iv) What do you think the best value of r is?

(c) $$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial^2 u(x,t)}{\partial t^2} \quad \text{for } 0 \le x \le 10,\ 0 \le t \le 10 \tag{P9.6.7}$$

with the initial/boundary conditions

$$u(x,0) = \begin{cases} (x-2)(3-x) & \text{for } 2 \le x \le 3 \\ 0 & \text{elsewhere} \end{cases} \tag{P9.6.8}$$

$$\partial u(x,t)/\partial t|_{t=0} = 0, \quad u(0,t) = 0, \quad u(10,t) = 0 \tag{P9.6.9}$$

(i) With the solution region divided into $M \times N = 100 \times 100$ sections, what is the value of $r = A(\Delta t)^2/(\Delta x)^2$?

(ii) Noting that the initial condition (P9.6.8) can be implemented by the MATLAB statement as

```
>>it0 = inline('(x-2).*(3-x).*(2<x&x<3)','x');
```

solve the PDE (P9.6.7) in the same way as in "`solve_wave.m`" and make a dynamic picture out of the numerical solution, with the current time printed on screen. Estimate the time when one of the two separated pulses propagating leftwards is reflected and reversed. How about the time when the two separated pulses are reunited?

9.7 FEM (Finite Element Method)

In expectation of better accuracy/resolution, modify the program "`do_fem.m`" (in Section 9.6) by appending the following lines

```
;-17/32 -31/64; -1/2 -17/32;-15/32 -31/64
;17/32 31/64; 1/2 17/32; 15/32 31/64
```

to the last part of the Node array `N` and replacing the last line of the subregion array S with

```
26 32 33; 27 33 34; 28 32 34; 29 35 36;
30 36 37; 31 35 37; 32 33 34; 35 36 37
```

This is equivalent to refining the triangular mesh in the subregions nearest to the point charges at (0.5, 0.5) and (−0.5, −0.5) as depicted in Fig. P9.7. Plot the new solution obtained by running the modified program. You may have to change a statement of the program as follows.

```
f962 ='(norm([x y]+[0.5 0.5])<1e-3)-(norm([x y]-[0.5 0.5])<1e-3)';
```

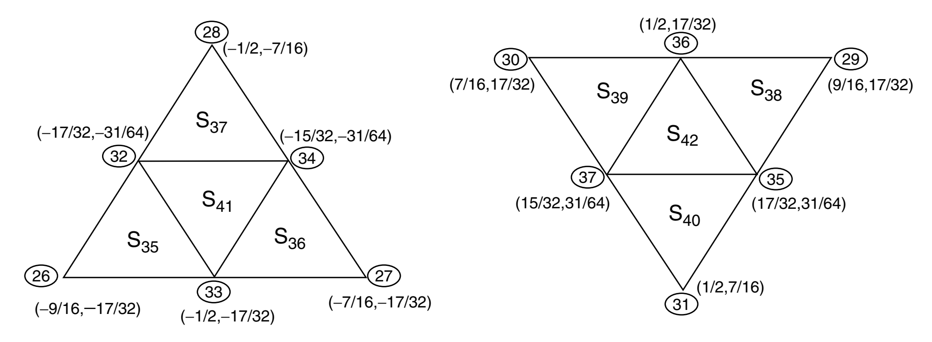


Figure P9.7 Refined triangular meshes.

9.8 PDEtool: GUI (Graphical User Interface) of MATLAB for Solving PDEs

(a) Consider the PDE

$$4\frac{\partial^2 u(x,y)}{\partial x^2} - 4\frac{\partial^2 u(x,y)}{\partial x \partial y} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0 \quad \text{for } 0 \le x \le 1, 0 \le y \le 1 \tag{P9.8.1}$$

with the boundary conditions

$$u(0,y) = ye^{2y}, \qquad u(1,y) = (1+y)e^{1+2y},$$
$$u(x,0) = xe^{x}, \qquad u(x,1) = (x+1)e^{x+2} \tag{P9.8.2}$$

Noting that the field of coefficient c should be filled in as

⊙ `Elliptic` ‖ `c` `4 -2 -2 1` `4 -2 1`

in the PDE specification dialog box and the true analytical solution is

$$u(x,y) = (x+y)e^{x+2y} \tag{P9.8.3}$$

use the PDEtool to solve this PDE and fill in Table P9.8.1 with the maximum absolute error and the number of nodes together with those of Problem 9.2(d) for comparison.

You can refer to Example 9.8 for the procedure to get the numerical value of the maximum absolute error. Notice that the number of nodes is the number of columns of p, which is obtained by clicking 'Export_Mesh' in the Mesh pull-down menu and then, clicking the OK button in the Export dialog box. You can also refer to Example 9.10 for the usage of 'Adaptive Mesh', but in this case you only have to check the box on the left of 'Adaptive Mode' and click the OK button in the 'Solve Parameters' dialog box opened by clicking 'Parameters' in the Solve pull-down menu, and then the mesh is adaptively refined every time you click the = button in the tool-bar to get the solution. With the box on the left of 'Adaptive Mode' unchecked in the 'Solve Parameters' dialog box,

Table P9.8.1 The Maximum Absolute Error and the Number of Nodes

	The Maximum Absolute Error	The Number of Nodes
poisson()	1.9256	41 × 41
PDEtool with Initialize Mesh	0.1914	177
PDEtool with Refine Mesh		
PDEtool with second Refine Mesh		
PDEtool with Adaptive Mesh		
PDEtool with second Adaptive Mesh		

the mesh is nonadaptively refined every time you click 'Refine Mesh' in the Mesh pull-down menu. You can restore the previous mesh by clicking 'Undo Mesh Change' in the Mesh pull-down menu.

(b) Consider the PDE

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0 \qquad \text{for } 0 \le x \le 4,\ 0 \le y \le 4 \quad \text{(P9.8.4)}$$

with the Dirichlet/Neumann boundary conditions

$$u(0,y) = e^y - \cos y,\ \ \partial u(x,y)/\partial x|_{x=4} = -e^y \sin 4 - e^4 \cos y \quad \text{(P9.8.5)}$$

$$\partial u(x,y)/\partial y|_{y=0} = \cos x,\ \ \partial u(x,y)/\partial y|_{y=4} = e^4 \cos x + e^x \sin 4 \quad \text{(P9.8.6)}$$

Noting that the true analytical solution is

$$u(x,y) = e^y \cos x - e^x \cos y \quad \text{(P9.8.7)}$$

use the PDEtool to solve this PDE and fill in Table P9.8.2 with the maximum absolute error and the number of nodes together with those of Problem 9.3(g) for comparison.

(c) Consider the PDE

$$2\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t} \qquad \text{for } 0 \le x \le \pi, \quad 0 \le t \le 0.2 \quad \text{(P9.8.8)}$$

with the initial/boundary conditions

$$u(x,0) = \sin(2x), \qquad u(0,t) = 0, \qquad u(\pi,t) = 0 \quad \text{(P9.8.9)}$$

Noting that the true analytical solution is

$$u(x,t) = \sin(2x)e^{-8t} \quad \text{(P9.8.10)}$$

Table P9.8.2 The Maximum Absolute Error and the Number of Nodes

	The Maximum Absolute Error	The Number of Nodes
poisson()	0.2005	21×21
PDEtool with Initialize Mesh	0.5702	177
PDEtool with Refine Mesh		
PDEtool with second Refine Mesh		
PDEtool with Adaptive Mesh		
PDEtool with second Adaptive Mesh		

Table P9.8.3 The Maximum Absolute Error and the Number of Nodes

	The Maximum Absolute Error	The Number of Nodes
poisson()	7.5462×10^{-4}	41×101
PDEtool with Initialize Mesh		
PDEtool with Refine Mesh		
PDEtool with second Refine Mesh		

use the PDEtool to solve this PDE and fill in Table P9.8.3 with the maximum absolute error and the number of nodes together with those obtained with the MATLAB routine '`heat_CN()`' in Problem 9.4(c) for comparison. In order to do this job, take the following steps.

(1) Click the □ button in the tool-bar and click-and-drag on the graphic region to create a rectangular domain. Then, double-click the rectangle to open the Object dialog box and set the Left/Bottom/Width/Height to 0/0/pi/0.01 to make a long rectangular domain.

(cf) Even if the PDEtool is originally designed to deal with only 2-D PDEs, we can use it to solve 1-D PDEs like (P9.8.8) by proceeding in this way.

(2) Click the $\partial\Omega$ button in the tool-bar, double-click the upper/lower boundary segments to set the homogeneous Neumann boundary condition ($g = 0, q = 0$) and double-click the left/right boundary segments to set the Dirichlet boundary condition ($h = 1, r = 0$) as given by Eq. (P9.8.9).

(3) Open the PDE specification dialog box by clicking the PDE button, check the box on the left of 'Parabolic' as the type of PDE, and set its parameters in Eq. (9.5.5) as $c = 2$, $a = 0$, $f = 0$ and $d = 1$, which corresponds to Eq. (P9.8.8).

(4) Click 'Parameters' in the Solve pull-down menu to set the time range, say, as 0:0.002:0.2 and to set the initial conditions as Eq. (P9.8.9).

(5) In the Plot selection dialog box opened by clicking the button, check the box before Height and click the Plot button. If you want to plot the solution graph at a time other than the final time, select the time for plot from $\{0, 0.002, 0.004, \ldots, 0.2\}$ in the far-right field of the Plot selection dialog box and click the Plot button again.

(6) If you want to see a movie-like dynamic picture of the solution graph, check the box before Animation and then click the Plot button in the Plot selection dialog box.

(7) Click 'Export_Mesh' in the Mesh pull-down menu, and then click the OK button in the Export dialog box to extract the mesh data {`p,e,t`}.

Also click ‘Export_Solution’ in the Solve pull-down menu, and then click the OK button in the Export dialog box to extract the solution u. Now, you can estimate how far the graphical/numerical solution deviates from the true solution (P9.8.10) by typing the following statements into the MATLAB command window:

```
>>x = p(1,:)'; y = p(2,: )'; %x,y coordinates of nodes in columns
>>tt = 0:0.01:0.2; %time vector in row
>>err = sin(2*x)*exp(-8*tt)-u; %deviation from true sol.(P9.8-10)
>>err_max = max(abs(err)) %maximum absolute error
```

(d) Consider the PDE

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial^2 u(x,t)}{\partial t^2} \quad \text{for } 0 \le x \le 10,\ 0 \le t \le 10 \qquad \text{(P9.8.11)}$$

with the initial/boundary conditions

$$u(x,0) = \begin{cases} (x-2)(3-x) & \text{for } 2 \le x \le 3 \\ 0 & \text{elsewhere} \end{cases}, \quad \frac{\partial u}{\partial t}\Big|_{t=0} = 0 \qquad \text{(P9.8.12)}$$

$$u(0,t) = 0, \qquad u(10,t) = 0 \qquad \text{(P9.8.13)}$$

Use the PDEtool to make a dynamic picture out of the solution for this PDE and see if the result is about the same as that obtained in Problem 9.6(c) in terms of the time when one of the two separated pulses propagating leftward is reflected and reversed and the time when the two separated pulses are reunited.

(cf) Even if the PDEtool is originally designed to solve only 2-D PDEs, we can solve 1-D PDE like (P9.8.11) by proceeding as follows:

(0) In the PDE toolbox window, adjust the ranges of the x axis and the y axis to [−0.5 10.5] and [−0.01 +0.01], respectively, in the box opened by clicking ‘Axes_Limits’ in the Options pull-down menu.

(1) Click the □ button in the tool-bar and click-and-drag on the graphic region to create a long rectangle of domain ranging from $x_0 = 0$ to $x_f = 10$. Then, double-click the rectangle to open the Object dialog box and set the Left/Bottom/Width/Height to 0/−0.01/10/0.02.

(2) Click the $\partial\Omega$ button in the tool-bar, double-click the upper/lower boundary segments to set the homogeneous Neumann boundary condition ($g = 0, q = 0$) and double-click the left/right boundary segments to set the Dirichlet boundary condition ($h = 1, r = 0$) as given by Eq. (P9.8.13).

(3) Open the PDE specification dialog box by clicking the PDE button, check the box on the left of ‘Hyperbolic’ as the type of PDE, and set its parameters in Eq. (P9.8.11) as $c = 1, a = 0, f = 0$ and $d = 1$. (See Fig. 9.15a.)

(4) Click ‘Parameters’ in the Solve pull-down menu to set the time range to, say, as 0:0.2:10, the boundary condition as (P9.8.13) and the initial conditions as (P9.8.12). (See Fig. 9.15b and Problem 9.6(c)(ii).)

(5) In the Plot selection dialog box opened by clicking the button, check the box before ‘Height’ and the box before ‘Animation’ and then click the Plot button in the Plot selection dialog box to see a movie-like dynamic picture of the solution graph.

(6) If you want to have better resolution in the solution graph, click Mesh in the top menu bar and click ‘Refine Mesh’ in the Mesh pull-down menu. Then, select Plot in the top menu bar or type CTRL + P(^P) on the keyboard and click ‘Plot_Solution’ in the Plot pull-down menu to see a smoother animation graph.

(7) In order to estimate the time when one of the two separated pulses propagating leftward is reflected and reversed and the time when the two separated pulses are reunited, count the flickering frame numbers, noting that one flickering corresponds to 0.2 s according to the time range set in step (4).

(8) If you want to save the PDEtool program, click File in the top menu bar, click ‘Save_As’ in the File pull-down menu, and input the file name of your choice.

APPENDIX A

MEAN VALUE THEOREM

Theorem A.1. Mean Value Theorem[1]. Let a function $f(x)$ be continuous on the interval $[a, b]$ and differentiable over (a, b). Then, there exists at least one point ξ between a and b at which

$$f'(\xi) = \frac{f(b) - f(a)}{b - a}, \qquad f(b) = f(a) + f'(\xi)(b - a) \tag{A.1}$$

In other words, the curve of a continuous function $f(x)$ has the same slope as the straight line connecting the two end points $(a, f(a))$ and $(b, f(b))$ of the curve at some point $\xi \in [a, b]$, as in Fig. A.1.

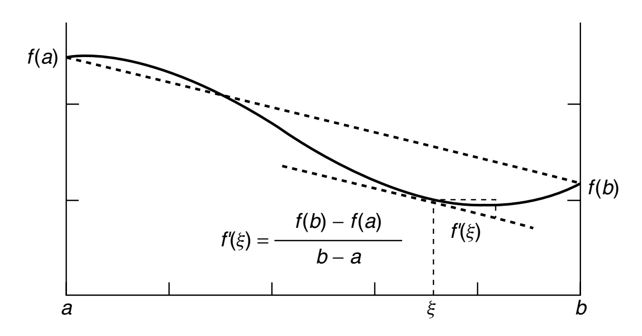


Figure A.1 Mean value theorem.

[1] See the website @http://www.maths.abdn.ac.uk/∼igc/testing/tch/ma2001/notes/notes.html

Theorem A.2. Taylor Series Theorem[1]. If a function $f(x)$ is continuous and its derivatives up to order $(K+1)$ are also continuous on an open interval D containing some point a, then the value of the function $f(x)$ at any point $x \in D$ can be represented by

$$f(x) = \sum_{k=0}^{K} \frac{f^k(a)}{k!}(x-a)^k + R_{K+1}(x) \tag{A.2}$$

where the first term of the right-hand side is called the Kth-degree Taylor polynomial, and the second term called the remainder (error) term is

$$R_{K+1}(x) = \frac{f^{(K+1)}(\xi)}{(K+1)!}(x-a)^{K+1} \qquad \text{for some } \xi \text{ between } a \text{ and } x \tag{A.3}$$

Moreover, if the function $f(x)$ has continuous derivatives of all orders on D, then the above representation becomes

$$f(x) = \sum_{k=0}^{\infty} \frac{f^k(a)}{k!}(x-a)^k \tag{A.4}$$

which is called the (infinite) Taylor series expansion of $f(x)$ about a.

APPENDIX B

MATRIX OPERATIONS/PROPERTIES

B.1 ADDITION AND SUBTRACTION

$$A + B = \begin{bmatrix} a_{11} & a_{12} & \cdot & a_{1N} \\ a_{21} & a_{22} & \cdot & a_{2N} \\ \cdot & \cdot & \cdot & \cdot \\ a_{M1} & a_{M2} & \cdot & a_{MN} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdot & b_{1N} \\ b_{21} & b_{22} & \cdot & b_{2N} \\ \cdot & \cdot & \cdot & \cdot \\ b_{M1} & b_{M2} & \cdot & b_{MN} \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} & c_{12} & \cdot & c_{1N} \\ c_{21} & c_{22} & \cdot & c_{2N} \\ \cdot & \cdot & \cdot & \cdot \\ c_{M1} & c_{M2} & \cdot & c_{MN} \end{bmatrix} = C \qquad \text{(B.1.1)}$$

with

$$a_{mn} + b_{mn} = c_{mn} \qquad \text{(B.1.2)}$$

B.2 MULTIPLICATION

$$AB = \begin{bmatrix} a_{11} & a_{12} & \cdot & a_{1K} \\ a_{21} & a_{22} & \cdot & a_{2K} \\ \cdot & \cdot & \cdot & \cdot \\ a_{M1} & a_{M2} & \cdot & a_{MK} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdot & b_{1N} \\ b_{21} & b_{22} & \cdot & b_{2N} \\ \cdot & \cdot & \cdot & \cdot \\ b_{K1} & b_{K2} & \cdot & b_{KN} \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} & c_{12} & \cdot & c_{1N} \\ c_{21} & c_{22} & \cdot & c_{2N} \\ \cdot & \cdot & \cdot & \cdot \\ c_{M1} & c_{M2} & \cdot & c_{MN} \end{bmatrix} = C \qquad \text{(B.2.1)}$$

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

with

$$c_{mn} = \sum_{k=1}^{K} a_{mk} b_{kn} \tag{B.2.2}$$

(cf) For this multiplication to be done, the number of columns of A must equal the number of rows of B.

(cf) Note that the commutative law does not hold for the matrix multiplication, that is, $AB \neq BA$.

B.3 DETERMINANT

The *determinant* of a $K \times K$ (square) matrix $A = [a_{mn}]$ is defined by

$$\det(A) = |A| = \sum_{k=0}^{K} a_{kn}(-1)^{k+n} M_{kn} \quad \text{or} \quad \sum_{k=0}^{K} a_{mk}(-1)^{m+k} M_{mk} \tag{B.3.1}$$

$$\text{for any fixed } 1 \leq n \leq K \text{ or } 1 \leq m \leq K$$

where the minor M_{kn} is the determinant of the $(K-1) \times (K-1)$ (minor) matrix formed by removing the kth row and the nth column from A and $A_{kn} = (-1)^{k+n} M_{kn}$ is called the cofactor of a_{kn}.

In particular, the determinants of a 2×2 matrix $A_{2\times 2}$ and a 3×3 matrix $A_{3\times 3}$ are

$$\det(A_{2\times 2}) = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = \sum_{k=1}^{2} a_{kn}(-1)^{k+n} M_{kn} = a_{11}a_{22} - a_{12}a_{21} \tag{B.3.2}$$

$$\det(A_{3\times 3}) = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

$$= a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22}a_{31}) \tag{B.3.3}$$

Note the following properties.

- If the determinant of a matrix is zero, the matrix is singular.
- The determinant of a matrix equals the product of the eigenvalues of a matrix.
- If A is upper/lower triangular having only zeros below/above the diagonal in each column, its determinant is the product of the diagonal elements.
- $\det(A^T) = \det(A)$; $\det(AB) = \det(A)\det(B)$; $\det(A^{-1}) = 1/\det(A)$

B.4 EIGENVALUES AND EIGENVECTORS OF A MATRIX[2]

The *eigenvalue* or characteristic value and its corresponding *eigenvector* or characteristic vector of an $N \times N$ matrix A are defined to be a scalar λ and a nonzero vector $\mathbf{v}$ satisfying

$$A\mathbf{v} = \lambda\mathbf{v} \Leftrightarrow (A - \lambda I)\mathbf{v} = \mathbf{0} \ (\mathbf{v} \neq \mathbf{0}) \tag{B.4.1}$$

where $(\lambda, \mathbf{v})$ is called an eigenpair and there are N eigenpairs for an $N \times N$ matrix A.

The eigenvalues of a matrix can be computed as the roots of the characteristic equation

$$|A - \lambda I| = 0 \tag{B.4.2}$$

and the eigenvector corresponding to an eigenvalue λ_i can be obtained by substituting λ_i into Eq. (B.4.1) and solve it for $\mathbf{v}$.

Note the following properties.

- If A is symmetric, all the eigenvalues are real-valued.
- If A is symmetric and positive definite, all the eigenvalues are real and positive.
- If $\mathbf{v}$ is an eigenvector of A, so is $c\mathbf{v}$ for any nonzero scalar c.

B.5 INVERSE MATRIX

The *inverse* matrix of a $K \times K$ (square) matrix $A = [a_{mn}]$ is denoted by A^{-1} and defined to be a matrix which is premultiplied/postmultiplied by A to form an identity matrix—that is, satisfies

$$A \times A^{-1} = A^{-1} \times A = I \tag{B.5.1}$$

An element of the inverse matrix $A^{-1} = [\alpha_{mn}]$ can be computed as

$$\alpha_{mn} = \frac{1}{\det(A)} A_{mn} = \frac{1}{|A|} (-1)^{m+n} M_{mn} \tag{B.5.2}$$

where M_{kn} is the minor of a_{kn} and $A_{kn} = (-1)^{k+n} M_{kn}$ is the cofactor of a_{kn}.

[2] See the website @http://www.sosmath.com/index.html or http://www.psc.edu/~burkardt/papers/linear_glossary.html.)

Note that a square matrix A is invertible/nonsingular if and only if

- No eigenvalue of A is zero, or equivalently,
- The rows (and the columns) of A are linearly independent, or equivalently,
- The determinant of A is nonzero.

B.6 SYMMETRIC/HERMITIAN MATRIX

A square matrix A is said to be *symmetric* if it is equal to its transpose, that is,

$$A^T \equiv A \tag{B.6.1}$$

A complex-valued matrix is said to be *Hermitian* if it is equal to its complex conjugate transpose, that is,

$$A \equiv A^{*T} \qquad \text{where } ^* \text{ means the conjugate.} \tag{B.6.2}$$

Note the following properties of a symmetric/Hermitian matrix.

- All the eigenvalues are real.
- If all the eigenvalues are distinct, the eigenvectors can form an orthogonal/unitary matrix U.

B.7 ORTHOGONAL/UNITARY MATRIX

A nonsingular (square) matrix A is said to be *orthogonal* if its transpose is equal to its inverse, that is,

$$A^T A \equiv I, \qquad A^T \equiv A^{-1} \tag{B.7.1}$$

A complex-valued (square) matrix is said to be *unitary* if its conjugate transpose is equal to its inverse, that is,

$$A^{*T} A \equiv I, \qquad A^{*T} \equiv A^{-1} \tag{B.7.2}$$

Note the following properties of an orthogonal/unitary matrix.

- The magnitude (absolute value) of every eigenvalue is one.
- The product of two orthogonal matrices is also orthogonal; $(AB)^{*T}(AB) = B^{*T}(A^{*T}A)B \equiv I$.

B.8 PERMUTATION MATRIX

A matrix P having only one nonzero element of value 1 in each row and column is called a *permutation* matrix and has the following properties.

- Premultiplication/postmultiplication of a matrix A by a permutation matrix P (i.e., PA or AP) yields the row/column change of the matrix A, respectively.
- A permutation matrix A is orthogonal, that is, $A^T A \equiv I$.

B.9 RANK

The *rank* of an $M \times N$ matrix is the number of linearly independent rows/columns and if it equals $\min(M, N)$, then the matrix is said to be of *maximal* or *full* rank; otherwise, the matrix is said to be *rank-deficient* or to have *rank-deficiency*.

B.10 ROW SPACE AND NULL SPACE

The *row space* of an $M \times N$ matrix A, denoted by $\mathcal{R}(\mathcal{A})$, is the space spanned by the row vectors—that is, the set of all possible linear combinations of row vectors of A that can be expressed by $A^T \boldsymbol{\alpha}$ with an M-dimensional column vector $\boldsymbol{\alpha}$. On the other hand, the *null space* of the matrix A, denoted by $\mathcal{N}(\mathcal{A})$, is the space orthogonal (perpendicular) to the row space—that is, the set of all possible linear combinations of the N-dimensional vectors satisfying $A\mathbf{x} = \mathbf{0}$.

B.11 ROW ECHELON FORM

A matrix is said to be of *row echelon form* if

- Each nonzero row having at least one nonzero element has a 1 as its first nonzero element.
- The leading 1 in a row is in a column to the right of the leading 1 in the upper row.
- All-zero rows are below the rows that have at least one nonzero element.

A matrix is said to be of *reduced row echelon form* if it satisfies the above conditions and, additionally, each column containing a leading 1 has no other nonzero elements.

Any matrix, singular or rectangular, can be transformed into this form through the Gaussian elimination procedure (i.e., a series of elementary row operations) or, equivalently, by using the MATLAB built-in routine "`rref()`". For example, we have

$$A = \begin{bmatrix} 0 & 0 & 1 & 3 \\ 2 & 4 & 0 & -8 \\ 1 & 2 & 1 & -1 \end{bmatrix} \xrightarrow[\text{change}]{\text{row}} \begin{bmatrix} 2 & 4 & 0 & -8 \\ 0 & 0 & 1 & 3 \\ 1 & 2 & 1 & -1 \end{bmatrix}$$

$$\xrightarrow[\substack{\text{row} \\ \text{subtraction}}]{\substack{\text{row} \\ \text{division}}} \begin{bmatrix} 1 & 2 & 0 & -4 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 1 & 3 \end{bmatrix} \xrightarrow[\text{subtraction}]{\text{row}} \begin{bmatrix} 1 & 2 & 0 & -4 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix} = rref(A)$$

Once this form is obtained, it is easy to compute the rank, the determinant and the inverse of the matrix, if only the matrix is invertible.

B.12 POSITIVE DEFINITENESS

A square matrix A is said to be *positive definite* if

$$\mathbf{x}^{*T} A\mathbf{x} > 0 \qquad \text{for any nonzero vector } \mathbf{x} \tag{B.12.1}$$

A square matrix A is said to be *positive semidefinite* if

$$\mathbf{x}^{*T} A\mathbf{x} \geq 0 \qquad \text{for any nonzero vector } \mathbf{x} \tag{B.12.2}$$

Note the following properties of a positive definite matrix A.

- A is nonsingular and all of its eigenvalues are positive.
- The inverse of A is also positive definite.

There are similar definitions for *negative definiteness* and *negative semidefiniteness*.

Note the following property, which can be used to determine if a matrix is positive (semi-) definite or not. A square matrix is positive definite if and only if:

(i) Every diagonal element is positive.
(ii) Every leading principal minor matrix has positive determinant.

On the other hand, a square matrix is positive semidefinite if and only if:

(i) Every diagonal element is nonnegative.
(ii) Every principal minor matrix has nonnegative determinant.

Note also that the principal minor matrices are the submatrices taking the diagonal elements from the diagonal of the matrix A and, say for a 3×3 matrix, the principal minor matrices are

$$a_{11}, a_{22}, a_{33}, \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix}, \begin{bmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{bmatrix}, \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

among which the leading ones are

$$a_{11}, \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

B.13 SCALAR (DOT) PRODUCT AND VECTOR (CROSS) PRODUCT

A *scalar product* of two N-dimensional vectors $\mathbf{x}$ and $\mathbf{y}$ is denoted by $\mathbf{x} \cdot \mathbf{y}$ and is defined by

$$\mathbf{x} \cdot \mathbf{y} = \sum_{n=1}^{N} x_n y_n = \mathbf{x}^T \mathbf{y} \tag{B.13.1}$$

An *outer product* of two three-dimensional column vectors $\mathbf{x} = [x_1 \ x_2 \ x_3]^T$ and $\mathbf{y} = [y_1 \ y_2 \ y_3]^T$ is denoted by $\mathbf{x} \times \mathbf{y}$ and is defined by

$$\mathbf{x} \times \mathbf{y} = \begin{bmatrix} x_2 y_3 - x_3 y_2 \\ x_3 y_1 - x_1 y_3 \\ x_1 y_2 - x_2 y_1 \end{bmatrix} \tag{B.13.2}$$

B.14 MATRIX INVERSION LEMMA

Matrix Inversion Lemma. Let A, C, and $[C^{-1} + DA^{-1}B]$ be well-defined with nonsingularity as well as compatible dimensions. Then we have

$$[A + BCD]^{-1} = A^{-1} - A^{-1}B[C^{-1} + DA^{-1}B]^{-1}DA^{-1} \tag{B.14.1}$$

Proof. We will show that postmultiplying Eq. (B.14.1) by $[A + BCD]$ yields an identity matrix.

$$\begin{aligned}
&[A^{-1} - A^{-1}B[C^{-1} + DA^{-1}B]^{-1}DA^{-1}][A + BCD] \\
&\quad = I + A^{-1}BCD - A^{-1}B[C^{-1} + DA^{-1}B]^{-1}D \\
&\qquad - A^{-1}B[C^{-1} + DA^{-1}B]^{-1}DA^{-1}BCD
\end{aligned}$$

$$= I + A^{-1}BCD - A^{-1}B[C^{-1} + DA^{-1}B]^{-1}C^{-1}CD$$
$$- A^{-1}B[C^{-1} + DA^{-1}B]^{-1}DA^{-1}BCD$$
$$= I + A^{-1}BCD - A^{-1}B[C^{-1} + DA^{-1}B]^{-1}[C^{-1} + DA^{-1}B]CD$$
$$= I + A^{-1}BCD - A^{-1}BCD \equiv I$$

APPENDIX C

DIFFERENTIATION WITH RESPECT TO A VECTOR

The first derivative of a scalar-valued function $f(\mathbf{x})$ with respect to a vector $\mathbf{x} = [x_1\ x_2]^T$ is called the gradient of $f(\mathbf{x})$ and defined as

$$\nabla f(\mathbf{x}) = \frac{d}{d\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \end{bmatrix} \tag{C.1}$$

Based on this definition, we can write the following equation.

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{x}^T \mathbf{y} = \frac{\partial}{\partial \mathbf{x}} \mathbf{y}^T \mathbf{x} = \frac{\partial}{\partial \mathbf{x}} (x_1 y_1 + x_2 y_2) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \mathbf{y} \tag{C.2}$$

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{x}^T \mathbf{x} = \frac{\partial}{\partial \mathbf{x}} (x_1^2 + x_2^2) = 2 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 2\mathbf{x} \tag{C.3}$$

Also with an $M \times N$ matrix A, we have

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{x}^T A \mathbf{y} = \frac{\partial}{\partial \mathbf{x}} \mathbf{y}^T A^T \mathbf{x} = A\mathbf{y} \tag{C.4a}$$

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{y}^T A \mathbf{x} = \frac{\partial}{\partial \mathbf{x}} \mathbf{x}^T A^T \mathbf{y} = A^T \mathbf{y} \tag{C.4b}$$

where

$$\mathbf{x}^T A \mathbf{y} = \sum_{m=1}^{M} \sum_{n=1}^{N} a_{mn} x_m y_n \tag{C.5}$$

Especially for a square, symmetric matrix A with $M = N$, we have

$$\frac{\partial}{\partial \mathbf{x}}\mathbf{x}^T A\mathbf{x} = (A + A^T)\mathbf{x} \xrightarrow{\text{if } A \text{ is symmetric}} 2A\mathbf{x} \tag{C.6}$$

The second derivative of a scalar function $f(\mathbf{x})$ with respect to a vector $\mathbf{x} = [x_1\ x_2]^T$ is called the Hessian of $f(\mathbf{x})$ and is defined as

$$H(\mathbf{x}) = \nabla^2 f(\mathbf{x}) = \frac{d^2}{d\mathbf{x}^2}f(\mathbf{x}) = \begin{bmatrix} \partial^2 f/\partial x_1^2 & \partial^2 f/\partial x_1 \partial x_2 \\ \partial^2 f/\partial x_2 \partial x_1 & \partial^2 f/\partial x_2^2 \end{bmatrix} \tag{C.7}$$

Based on this definition, we can write the following equation:

$$\frac{d^2}{d\mathbf{x}^2}\mathbf{x}^T A\mathbf{x} = A + A^T \xrightarrow{\text{if } A \text{ is symmetric}} 2A \tag{C.8}$$

On the other hand, the first derivative of a vector-valued function $\mathbf{f}(\mathbf{x})$ with respect to a vector $\mathbf{x} = [x_1\ x_2]^T$ is called the Jacobian of $f(\mathbf{x})$ and is defined as

$$J(\mathbf{x}) = \frac{d}{d\mathbf{x}}\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{bmatrix} \tag{C.9}$$

APPENDIX D

LAPLACE TRANSFORM

Table D.1 Laplace Transforms of Basic Functions

$x(t)$	$X(s)$	$x(t)$	$X(s)$	$x(t)$	$X(s)$
(1) $\delta(t)$	1	(5) $e^{-at}u_s(t)$	$\frac{1}{s+a}$	(9) $e^{-at}\sin\omega t\ u_s(t)$	$\frac{\omega}{(s+a)^2+\omega^2}$
(2) $\delta(t-t_1)$	$e^{-t_1 s}$	(6) $t^m e^{-at}u_s(t)$	$\frac{m!}{(s+a)^{m+1}}$	(10) $e^{-at}\cos\omega t\ u_s(t)$	$\frac{s+a}{(s+a)^2+\omega^2}$
(3) $u_s(t)$	$\frac{1}{s}$	(7) $\sin\omega t\ u_s(t)$	$\frac{\omega}{s^2+\omega^2}$		
(4) $t^m u_s(t)$	$\frac{m!}{s^{m+1}}$	(8) $\cos\omega t\ u_s(t)$	$\frac{s}{s^2+\omega^2}$		

Table D.2 Properties of Laplace Transform

Property	Expression
(0) Definition	$X(s) = L\{x(t)\} = \int_0^{\infty} x(t)e^{-st}dt$
(1) Linearity	$\alpha x(t) + \beta x(t) \rightarrow \alpha X(s) + \beta Y(s)$
(2) Time shifting	$x(t - t_1)u_s(t - t_1), t_1 > 0 \rightarrow e^{-st_1}\left\{X(s) + \int_{-t_1}^{0} x(\tau)e^{-s\tau}\,d\tau\right\}$
(3) Frequency shifting	$e^{s_1 t}x(t) \rightarrow X(s - s_1)$
(4) Real convolution	$g(t) * x(t) \rightarrow G(s)X(s)$
(5) Time derivative	$x'(t) \rightarrow sX(s) - x(0)$
(6) Time integral	$\int_{-\infty}^{t} x(\tau)\,d\tau \rightarrow \frac{1}{s}X(s) + \frac{1}{s}\int_{-\infty}^{0} x(\tau)\,d\tau$
(7) Complex derivative	$t\ x(t) \rightarrow -\frac{d}{ds}X(s)$
(8) Complex convolution	$x(t)y(t) \rightarrow \frac{1}{2\pi j}\int_{\sigma_0-\infty}^{\sigma_0+\infty} X(v)Y(s - v)\,dv$
(9) Initial value theorem	$x(0) \rightarrow \lim_{s\rightarrow\infty} sX(s)$
(10) Final value theorem	$x(\infty) \rightarrow \lim_{s\rightarrow 0} sX(s)$

APPENDIX E

FOURIER TRANSFORM

Table E.1 Properties of CtFT (Continuous-Time Fourier Transform)

(0) Definition	$X(\omega) = F\{x(t)\} = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}\,dt$
(1) Linearity	$\alpha x(t) + \beta x(t) \to \alpha X(\omega) + \beta Y(\omega)$
(2) Symmetry	$x(t) = x_e(t) + x_o(t)$: real $\to X(\omega) \equiv X^*(-\omega)$
	$x_e(t)$: real and even $\to X_e(\omega) = \text{Re}\{X(\omega)\}$
	$x_o(t)$: real and odd $\to X_o(\omega) = j\text{Im}\{X(\omega)\}$
	$x(-t) \to X(-\omega)$
(3) Time shifting	$x(t - t_1) \to e^{-j\omega t_1}X(\omega)$
(4) Frequency shifting	$e^{j\omega_1 t}x(t) \to X(\omega - \omega_1)$
(5) Real convolution	$g(t) * x(t) = \int_{-\infty}^{\infty} g(\tau)x(t-\tau)\,d\tau \to G(\omega)X(\omega)$
(6) Time derivative	$x'(t) \to j\omega X(\omega)$
(7) Time integral	$\int_{-\infty}^{t} x(\tau)\,d\tau \to \frac{1}{j\omega}X(\omega) + \pi X(0)\delta(\omega)$
(8) Complex derivative	$t\ x(t) \to j\frac{d}{d\omega}X(\omega)$
(9) Complex convolution	$x(t)y(t) \to \frac{1}{2\pi}X(\omega) * Y(\omega)$
(10) Scaling	$x(at) \to \frac{1}{\lvert a\rvert}X(\omega/a)$
(11) Duality	$g(t) \to f(\omega) \Leftrightarrow f(t) \to 2\pi g(\omega)$
(12) Parseval's relation	$\int_{-\infty}^{\infty} \lvert x(t)\rvert^2\,dt \to \frac{1}{2\pi}\int_{-\infty}^{\infty} \lvert x(\omega)\rvert^2\,d\omega$

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

Table E.2 Properties of DtFT (Discrete-Time Fourier Transform)

Property	Expression
(0) Definition	$X(\Omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\Omega n}$
(1) Linearity	$\alpha x[n] + \beta x[n] \rightarrow \alpha X(\Omega) + \beta Y(\Omega)$
(2) Symmetry	$x[n] = x_e[n] + x_o[n]$: real $\rightarrow X(\Omega) \equiv X^*(-\Omega)$
	$x_e[n]$: real and even $\rightarrow X_e(\Omega) = \text{Re}\{X(\Omega)\}$
	$x_o[n]$: real and odd $\rightarrow X_o(\Omega) = j\text{Im}\{X(\Omega)\}$
	$x[-n] \rightarrow X(-\Omega)$
(3) Time shifting	$x[n-n_1] \rightarrow e^{-j\Omega n_1}X(\Omega)$
(4) Frequency shifting	$e^{j\Omega_1 n}x[n] \rightarrow X(\Omega - \Omega_1)$
(5) Real convolution	$g[n] * x[n] = \sum_{m=-\infty}^{\infty} g[m]x[n-m] \rightarrow G(\Omega)X(\Omega)$
(6) Complex derivative	$n\ x[n] \rightarrow j\frac{d}{d\Omega}X(\Omega)$
(7) Complex convolution	$x[n]y[n] \rightarrow \frac{1}{2\pi}X(\Omega) * Y(\Omega)$ (periodic/circular convolution)
(8) Scaling	$\begin{cases} x[n/M] & \text{if } n = mM (m: \text{an integer}) \\ 0, & \text{otherwise} \end{cases} \rightarrow X(M\Omega)$
(9) Parseval's relation	$\sum_{n=-\infty}^{\infty} \lvert x[n]\rvert^2 = \frac{1}{2\pi}\int_{2\pi} \lvert x(\Omega)\rvert^2 d\Omega$

APPENDIX F

USEFUL FORMULAS

Formulas for Summation of Finite Number of Terms

$$\sum_{n=0}^{N} a^n = \frac{1-a^{N+1}}{1-a} \quad \text{(F.1)}$$

$$\sum_{n=0}^{N} na^n = a\frac{1-(N+1)a^N + Na^{N+1}}{(1-a)^2} \quad \text{(F.2)}$$

$$\sum_{n=0}^{N} n = \frac{N(N+1)}{2} \quad \text{(F.3)}$$

$$\sum_{n=0}^{N} n^2 = \frac{N(N+1)(2N+1)}{6} \quad \text{(F.4)}$$

$$\sum_{n=0}^{N} n(n+1) = \frac{N(N+1)(N+2)}{3} \quad \text{(F.5)}$$

$$(a+b)^N = \sum_{n=0}^{N} N\mathrm{C}_n a^{N-n} b^n \text{ with } N\mathrm{C}_n = N\mathrm{C}_{N-n} = \frac{N\mathrm{P}_n}{n!} = \frac{N!}{(N-n)!n!} \quad \text{(F.6)}$$

Formulas for Summation of Infinite Number of Terms

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}, \quad |x| < 1 \quad \text{(F.7)}$$

$$\sum_{n=0}^{\infty} nx^n = \frac{x}{(1-x)^2}, \quad |x| < 1 \quad \text{(F.8)}$$

$$\sum_{n=0}^{\infty} n^k x^n = \lim_{a\to 0} (-1)^k \frac{\partial^k}{\partial a^k} \left\{ \frac{x}{x-e^{-a}} \right\}, \quad |x| < 1 \quad \text{(F.9)}$$

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \frac{1}{4}\pi \quad \text{(F.10)}$$

(*continued overleaf*)

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

$$\sum_{n=0}^{\infty} \frac{1}{n^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots = \frac{1}{6}\pi^2 \tag{F.11}$$

$$e^x = \sum_{n=0}^{\infty} \frac{1}{n!} x^n = 1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \cdots \tag{F.12}$$

$$a^x = \sum_{n=0}^{\infty} \frac{(\ln a)^n}{n!} x^n = 1 + \frac{\ln a}{1!}x + \frac{(\ln a)^2}{2!}x^2 + \frac{(\ln a)^3}{3!}x^3 + \cdots \tag{F.13}$$

$$\ln(1 \pm x) = -\sum_{n=1}^{\infty} (\pm 1)^n \frac{1}{n} x^n = \pm x - \frac{1}{2}x^2 \pm \frac{1}{3}x^3 - \cdots, |x| < 1 \tag{F.14}$$

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \cdots \tag{F.15}$$

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots \tag{F.16}$$

$$\tan x = x + \frac{1}{3}x^3 + \frac{2}{15}x^5 + \cdots, \qquad |x| < \frac{\pi}{2} \tag{F.17}$$

$$\tan^{-1} x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \cdots \tag{F.18}$$

Trigonometric Formulas

$$\sin(A \pm B) = \sin A \cos B \pm \cos A \sin B \tag{F.19}$$

$$\tan(A \pm B) = \frac{\tan A \pm \tan B}{1 \mp \tan A \tan B} \tag{F.21}$$

$$\cos(A \pm B) = \cos A \cos B \mp \sin A \sin B \tag{F.20}$$

$$\sin A \sin B = \frac{1}{2}\{\cos(A - B) - \cos(A + B)\} \tag{F.22}$$

$$\sin A \cos B = \frac{1}{2}\{\sin(A + B) + \sin(A - B)\} \tag{F.23}$$

$$\cos A \sin B = \frac{1}{2}\{\sin(A + B) - \sin(A - B)\} \tag{F.24}$$

$$\cos A \cos B = \frac{1}{2}\{\cos(A + B) + \cos(A - B)\} \tag{F.25}$$

$$\sin A + \sin B = 2 \sin\left(\frac{A + B}{2}\right) \cos\left(\frac{A - B}{2}\right) \tag{F.26}$$

$$\cos A + \cos B = 2 \cos\left(\frac{A + B}{2}\right) \cos\left(\frac{A - B}{2}\right) \tag{F.27}$$

$$a \cos A - b \sin A = \sqrt{a^2 + b^2} \cos(A + \theta), \theta = \tan^{-1}\left(\frac{b}{a}\right) \tag{F.28}$$

$$a \sin A + b \cos A = \sqrt{a^2 + b^2} \sin(A + \theta), \theta = \tan^{-1}\left(\frac{b}{a}\right) \tag{F.29}$$

$$\sin^2 A = \frac{1}{2}(1 - \cos 2A) \tag{F.30}$$

$$\cos^2 A = \frac{1}{2}(1 + \cos 2A) \tag{F.31}$$

$$\sin^3 A = \frac{1}{4}(3\sin A - \sin 3A) \quad \text{(F.32)}$$

$$\cos^3 A = \frac{1}{4}(3\cos A + \cos 3A) \quad \text{(F.33)}$$

$$\sin 2A = 2\sin A\cos A \quad \text{(F.34)}$$

$$\sin 3A = 3\sin A - 4\sin^3 A \quad \text{(F.35)}$$

$$\cos 2A = \cos^2 A - \sin^2 A = 1 - 2\sin^2 A = 2\cos^2 A - 1 \quad \text{(F.36)}$$

$$\cos 3A = 4\cos^3 A - 3\sin A \quad \text{(F.37)}$$

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C} \quad \text{(F.38)}$$

$$e^{\pm j\theta} = \cos\theta \pm j\sin\theta \quad \text{(F.40)}$$

$$a^2 = b^2 + c^2 - 2bc\cos A \quad \text{(F.39a)}$$

$$\sin\theta = \frac{1}{j2}(e^{j\theta} - e^{-j\theta}) \quad \text{(F.41a)}$$

$$b^2 = c^2 + a^2 - 2ca\cos B \quad \text{(F.39b)}$$

$$\cos\theta = \frac{1}{2}(e^{j\theta} + e^{-j\theta}) \quad \text{(F.41b)}$$

$$c^2 = a^2 + b^2 - 2ab\cos C \quad \text{(F.39c)}$$

$$\tan\theta = \frac{1}{j}\frac{e^{j\theta} - e^{-j\theta}}{e^{j\theta} + e^{-j\theta}} \quad \text{(F.41c)}$$

APPENDIX G

SYMBOLIC COMPUTATION

G.1 HOW TO DECLARE SYMBOLIC VARIABLES AND HANDLE SYMBOLIC EXPRESSIONS

To declare any variable(s) as a symbolic variable, you should use the `sym` or `syms` command as below.

```
>>a = sym('a'); t = sym('t'); x = sym('x');
>>syms a x y t %or, equivalently and more efficiently
```

Once the variables have been declared as symbolic, they can be used in expressions and as arguments to many functions without being evaluated as numeric.

```
>>f = x^2/(1 + tan(x)^2);
>>ezplot(f,-pi,pi)
>>simplify(cos(x)^2+sin(x)^2) %simplify an expression
   ans = 1
>>simplify(cos(x)^2 - sin(x)^2) %simplify an expression
   ans = 2*cos(x)^2-1
>>simple(cos(x)^2 - sin(x)^2) %simple expression
   ans = cos(2*x)
>>simple(cos(x) + i*sin(x)) %simple expression
   ans = exp(i*x)
>>eq1 = expand((x + y)^3 - (x + y)^2) %expand
  eq1 = x^3 + 3*x^2*y + 3*x*y^2 + y^3 - x^2 - 2*x*y - y^2
>>collect(eq1,y) %collect similar terms in descending order with respect to y
   ans = y^3 + (3*x - 1)*y^2 + (3*x^2 - 2*x)*y + x^3 - x^2
```

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

```
>>factor(eq1) %factorize
   ans = (x + y - 1)*(x + y)^2
>>horner(eq1) %nested multiplication form
   ans = (-1 + y)*y^2 + ((- 2 + 3*y)*y + (-1 + 3*y + x)*x)*x
>>pretty(ans) %pretty form
             2
   (-1 + y) y  + ((-2 + 3 y) y + (-1 + 3 y + x) x) x
```

If you need to substitute numeric values or other expressions for some symbolic variables in an expression, you can use the subs function as below.

```
>>subs(eq1,x,0) %substitute numeric value
   ans = -y^2 + y^3
>>subs(eq1,{x,y},{0,x - 1}) %substitute numeric values
   ans = (x - 1)^3 - (x - 1)^2
```

The sym command allows you to declare symbolic real variables by using the ‘real’ option as illustrated below.

```
>>x = sym('x','real'); y = sym('y','real');
>>syms x y real %or, equivalently
>>z = x + i*y; %declare z as a symbolic complex variable
>>conj(z) %complex conjugate
   ans = x - i*y
>>abs(z)
   ans = (x^2 + y^2)^(1/2) %equivalently
```

The sym function can be used to convert numeric values into their symbolic expressions.

```
>>sym(1/2) + 0.2
   ans = 7/10 %symbolic expression
```

On the other hand, the double command converts symbolic expressions into their numeric (double-precision floating-point) values and the vpa command finds the variable-precision arithmetic (VPA) expression (as a symbolic representation) of a numeric or symbolic expression with d significant decimal digits, where d is the current setting of DIGITS that can be set by the digits command. Note that the output of the vpa command is a symbolic expression even if it may look like a numeric value. Let us see some examples.

```
>>f = sym('exp(i*pi/4)')
    f = exp(i*pi/4)
>>double(f)
   ans = 0.7071 + 0.7071i %numeric value
>>vpa(ans,2)
   ans = .71 + .71*i %symbolic expression with 2 significant digits
```

G.2 CALCULUS

G.2.1 Symbolic Summation

We can use the `symsum()` function to obtain the sum of an indefinite/definite series as below.

```
>>syms x n N %declare x,n,N as symbolic variables
>>simple(symsum(n,0,N))
   ans = 1/2*N*(N + 1) %∑_{n=0}^N n = N(N + 1)/2
>>simple(symsum(n^2,0,N))
   ans = 1/6*N*(N + 1)*(2*N + 1) %∑_{n=0}^N n^2 = N(N + 1)(2N + 1)/6
>>symsum(1/n^2,1,inf))
   ans = 1/6*pi^2 %∑_{n=0}^N 1/n^2 = π^2/6
>>symsum(x^n,n,0,inf))
   ans = -1/(-1 + x) %∑_{n=0}^N x^n = 1/(1 - x) under the assumption that |x| < 1
```

G.2.2 Limits

We can use the `limit()` function to get the (two-sided) limit and the right/left-sided limits of a function as below.

```
>>syms h n x
>>limit(sin(x)/x,x,0) % lim_{x→0} sin x / x = 1
   ans = 1

>>limit(x/abs(x),x,0,'right') % lim_{x→0+} x/|x| = 1
   ans = 1

>>limit(x/abs(x),x,0,'left') % lim_{x→0-} x/|x| = -1
   ans = -1

>>limit(x/abs(x),x,0) % lim_{x→0} x/|x| = ?
   ans = NaN  %Not a Number

>>limit((cos(x+h)-cos(x))/h,h,0) % lim_{h→0} (cos(x + h) - cos(x))/h = d/dx cosx = -sinx
   ans = -sin(x)

>>limit((1 + x/n)^n,n,inf) % lim_{n→∞} (1 + x/n)^n = e^x
   ans = exp(x)
```

G.2.3 Differentiation

The `diff()` function differentiates a symbolic expression w.r.t. the variable given as one of its 2nd or 3rd input arguments or its free variable which might be determined by using the `findsym` function.

```
>>syms a b x n t
>>diff(x^n))
   ans = x^n*n/x
>>simplify(ans)
   ans = x^(n - 1)*n
>>f = exp(a*x)*cos(b*t)
>>diff(f) %equivalently diff(f,x)
   ans = a*exp(a*x)*cos(b*t) %
```
$\frac{d}{dx}f = \frac{d}{dx}e^{ax}\cos(bt) = ae^{ax}\cos(bt)$

```
>>diff(f,t)
   ans = -exp(a*x)*sin(b*t)*b %
```
$\frac{d}{dt}f = \frac{d}{dt}e^{ax}\cos(bt) = -be^{ax}\sin(bt)$

```
>>diff(f,2) %equivalently diff(f,x,2)
   ans = a^2*exp(a*x)*cos(b*t) %
```
$\frac{d^2}{dx^2}f = a^2e^{ax}\cos(bt)$

```
>>diff(f,t,2)
   ans = -exp(a*x)*cos(b*t)*b^2 %
```
$\frac{d^2}{dt^2}f = -e^{ax}\cos(bt)b^2$

```
>>g = [cos(x)*cos(t) cos(x)*sin(t)];
>>jacob_g = jacobian(g,[x t])
   jacob_g = [ -sin(x)*cos(t), -cos(x)*sin(t)]
             [ -sin(x)*sin(t),  cos(x)*cos(t)]
```

Note that the `jacobian()` function finds the jacobian defined by (C.9)—that is, the derivative of a vector function $[g_1 \quad g_2]^T$ with respect to a vector variable $[x \quad t]^T$—as

$$J = \begin{bmatrix} \partial g_1/\partial x & \partial g_1/\partial t \\ \partial g_2/\partial x & \partial g_2/\partial t \end{bmatrix} \tag{G.1}$$

G.2.4 Integration

The `int()` function returns the indefinite/definite integral (anti-derivative) of a function or an expression with respect to the variable given as its second input argument or its free variable which might be determined by using the `findsym` function.

```
>>syms a x y t
>>int(x^n)
   ans = x^(n + 1)/(n + 1) %
```
$\int x^n\,dx = \frac{1}{n+1}x^{n+1}$

```
>>int(1/(1 + x^2))
   ans = atan(x) %
```
$\int \frac{1}{1+x^2}\,dx = \tan^{-1}x$

```
>>int(a^x) %equivalently diff(f,x,2)
   ans  = 1/log(a)*a^x  %
```
$\int a^x\,dx = \frac{1}{\log a}a^x$

```
>>int(sin(a*t),0,pi) %equivalently int(sin(a*t),t,0,pi)
   ans = -cos(pi*a)/a + 1/a %
```
$\int_0^{\pi}\sin(at)\,dt = -\frac{1}{a}\cos(at)\Big|_0^{\pi} = -\frac{1}{a}\cos(a\pi) + \frac{1}{a}$

```
>>int(exp(-(x - a)^2),a,inf) %equivalently int(exp(-(x - a)^2),x,0,inf)
   ans = 1/2*pi^(1/2) % ∫_a^∞ e^-(x-a)^2 dx = ∫_0^∞ e^-x^2 dx = (1/2)√π
```

G.2.5 Taylor Series Expansion

We can use the `taylor()` function to find the Taylor series expansion of a function or an expression with respect to the variable given as its second or third input argument or its free variable that might be determined by using the `findsym` function.

One may put ‘`help taylor`’ into the MATLAB command window to see its usage, which is restated below. Let us try applying it.

```
>>syms x t; N = 3;
>>Tx0 = taylor(exp(-x),N + 1) %f(x) ≅ Σ_{n=0}^N (1/n!) f^(n)(0) x^n
   Tx0 = 1-x + 1/2*x^2 - 1/6*x^3

>>sym2poly(Tx0) %extract the coefficients of Taylor series polynomial
   ans = -0.1667    0.5000   -1.0000    1.0000

>>xo = 1; Tx1 = taylor(exp(-x),N + 1,xo) %f(x) ≅ Σ_{n=0}^N (1/n!) f^(n)(x0) (x - x0)^n
   Tx1 = exp(-1) - exp(-1)*(x - 1) + 1/2*exp(-1)*(x - 1)^2 - 1/6*exp(-1)*(x - 1)^3

>>pretty(Tx1)
                                                 2                    3
   exp(-1) -exp(-1)(x - 1) +1/2 exp(-1)(x - 1) -1/6 exp(-1)(x - 1)

>>f = exp(-x)*sin(t);
>>Tt = taylor(f,N + 1,t) %f(x) ≅ Σ_{n=0}^N (1/n!) f^(n)(0) t^n
  Tt = exp(-x)*t - 1/6*exp(-x)*t^3
```

- `taylor(f)` gives the fifth-order Maclaurin series expansion of `f`.
- `taylor(f,n+1)` with an integer `n > 0` gives the nth-order Maclaurin series expansion of `f`.
- `taylor(f,a)` with a real number (a) gives the fifth-order Taylor series expansion of `f` about `a`.
- `taylor(f,n + 1,a)` gives the nth-order Taylor series expansion of `f` about default_variable=a.
- `taylor(f,n + 1,a,y)` gives the nth-order Taylor series expansion of `f(y)` about `y = a`.

(cf) The target function `f` must be a legitimate expression given directly as the first input argument.

(cf) Before using the command “`taylor()`”, one should declare the arguments of the function as symbols by putting, say, “`syms x t`”.

(cf) In case the function has several arguments, it is a good practice to put the independent variable as the last input argument of “`taylor()`”, though `taylor()` takes

one closest (alphabetically) to ‘x’ as the independent variable by default only if it has been declared as a symbolic variable and is contained as an input argument of the function f.

(cf) One should use the MATLAB command “sym2poly()” if he wants to extract the coefficients from the Taylor series expansion obtained as a symbolic expression.

G.3 LINEAR ALGEBRA

Several MATLAB commands and functions can be used to manipulate the vectors or matrices consisting of symbolic expressions as well as those consisting of numerics.

```
>>syms a11 a12 a21 a22
>>A = [a11 a12; a21 a22];
>>det(A)
   ans  = a11*a22 - a12*a21
>>AI = A^ - 1
   AI = [  a22/(a11*a22 - a12*a21), -a12/(a11*a22 - a12*a21)]
        [ -a21/(a11*a22 - a12*a21),  a11/(a11*a22 - a12*a21)]
>>A*AI
   ans = [ a11*a22/(a11*a22 - a12*a21)-a12*a21/(a11*a22 - a12*a21),         0]
         [        0, a11*a22/(a11*a22 - a12*a21) - a12*a21/(a11*a22 - a12*a21)]
>>simplify(ans) %simplify an expression
   ans = [ 1,  0]
         [ 0,  1]
>>syms x t;
>>G = [cos(t) sin(t); -sin(t) cos(t)] %The Givens transformation matrix
   G = [  cos(t),  sin(t)]
       [ -sin(t),  cos(t)]
>>det(G), simple(ans)
   ans = cos(t)^2 + sin(t)^2
   ans = 1
>>G2 = G^2, simple(G2)
  G2 = [ cos(t)^2 - sin(t)^2,     2*cos(t)*sin(t)]
       [    -2*cos(t)*sin(t), cos(t)^2 - sin(t)^2]
  ans = [  cos(2*t),  sin(2*t)]
        [ -sin(2*t),  cos(2*t)]
>>GTG = G.'*G, simple(GTG)
   GTG = [ cos(t)^2 + sin(t)^2,                   0]
         [                   0, cos(t)^2 + sin(t)^2]
   ans = [ 1, 0]
         [ 0, 1]
>>simple(G^ - 1) %inv(G) for the inverse of Givens transformation matrix
   G = [  cos(t), -sin(t)]
       [  sin(t),  cos(t)]
>>syms b c
>>A = [0 1; -c -b];
>>[V,E] = eig(A)
   V = [ -(1/2*b + 1/2*(b^2 - 4*c)^(1/2))/c, -(1/2*b - 1/2*(b^2 - 4*c)^(1/2))/c]
       [                                  1,                                  1]
   E = [ -1/2*b + 1/2*(b^2 - 4*c)^(1/2),                              0]
       [                              0, -1/2*b - 1/2*(b^2 - 4*c)^(1/2)]
>> solve(poly(A))%another way to get eigenvalues(characteristic roots)
    ans = [ -1/2*b+1/2*(b^2 - 4*c)^(1/2)]
          [ -1/2*b-1/2*(b^2 - 4*c)^(1/2)]
```

Besides, other MATLAB functions such as `jordan(A)` and `svd(A)` can be used to get the Jordan canonical form together with the corresponding similarity transformation matrix and the singular value decomposition of a symbolic matrix.

G.4 SOLVING ALGEBRAIC EQUATIONS

We can use the backslash (\) operator to solve a set of linear equations written in a matrix–vector form.

```
>>syms R11 R12 R21 R22 b1 b2
>>R = [R11 R12; R21 R22]; b = [b1; b2];
>>x = R\b
  x = [  (R12*b2 - b1*R22)/(-R11*R22 + R21*R12)]
      [ (-R11*b2 + R21*b1)/(-R11*R22 + R21*R12)]
```

We can also use the MATLAB function `solve()` to solve symbolic algebraic equations.

```
>>syms a b c x
>>fx = a*x^2+b*x+c;
>>solve(fx) %formula for roots of 2nd-order polynomial eq
   ans = [ 1/2/a*(-b + (b^2 - 4*a*c)^(1/2))]
         [ 1/2/a*(-b - (b^2 - 4*a*c)^(1/2))]
>>syms x1 x2 b1 b2
>>fx1 = x1 + x2 - b1; fx2 = x1 + 2*x2 - b2; %a system of simultaneous algebraic eq.
>>[x1o,x2o] = solve(fx1,fx2) %
   x1o = 2*b1 - b2
   x2o = -b1 + b2
```

G.5 SOLVING DIFFERENTIAL EQUATIONS

We can use the MATLAB function `dsolve()` to solve symbolic differential equations.

```
>>syms a b c x
>>xo = dsolve('Dx + a*x = 0') % a differential eq.(d.e.) w/o initial condition
   xo = exp(-a*t)*C1 % a solution with undetermined constant
>>xo = dsolve('Dx + a*x = 0','x(0) = 2') % a d.e. with initial condition
   xo = 2*exp(-a*t) % a solution with undetermined constant
>>xo = dsolve('Dx=1+x^2') % a differential eq. w/o initial condition
   xo = tan(t - C1) % a solution with undetermined constant
>>xo = dsolve('Dx = 1 + x^2','x(0) = 1') % with the initial condition
   xo = tan(t + 1/4*pi) % a solution with determined constant
>>yo = dsolve('D2u = -u','t') % a 2nd-order d.e. without initial condition
   yo = C1*sin(t) + C2*cos(t)
>>xo = dsolve('D2u = -u','u(0) = 1,Du(0) = 0','t') % with the initial condition
   xo = cos(t))
>>yo = dsolve('(Dy)^2 + y^2 = 1','y(0) = 0','x') % a 1st-order nonlinear d.e.(nlde)
   yo = [  sin(x)]    %two solutions
        [ -sin(x)]
>>yo = dsolve('D2y = cos(2*x) - y','y(0) = 1,Dy(0) = 0','x') % a 2md-order nlde
   yo = 4/3*cos(x) - 2/3*cos(x)^2 + 1/3
>>S = dsolve('Df=3*f + 4*g','Dg=-4*f + 3*g');
```

```
>>f = S.f, g = S.g
    f = exp(3*t)*(C1*sin(4*t) + C2*cos(4*t))
    g = exp(3*t)*(C1*cos(4*t) - C2*sin(4*t))
>>[f,g] = dsolve('Df = 3*f + 4*g,Dg = -4*f + 3*g','f(0) = 0,g(0) = 1')
    f = exp(3*t)*sin(4*t)
    g = exp(3*t)*cos(4*t)
```

APPENDIX H

SPARSE MATRICES

A matrix is said to be sparse if it has a large portion of zero elements. MATLAB has some built-in functions/routines that enable us to exploit the sparsity of a matrix for computational efficiency.

The MATLAB routine `sparse()` can be used to convert a (regular) matrix into a sparse form by squeezing out any zero elements and to generate a sparse matrix having the elements of a vector given together with the row/column index vectors. On the other hand, the MATLAB routine `full()` can be used to convert a matrix of sparse form into a regular one.

```
>>row_index = [1 1 2 3 4]; col_index = [1 2 2 3 4]; elements = [1 2 3 4 5];
>>m = 4; n = 4; As = sparse(row_index,col_index,elements,m,n)
    As = (1,1)        1
         (1,2)        2
         (2,2)        3
         (3,3)        4
         (4,4)        5
>>Af = full(As)
   Af =    1     2     0     0
           0     3     0     0
           0     0     4     0
           0     0     0     5
```

We can use the MATLAB routine `sprandn(m,n,nzd)` to generate an m × n sparse matrix having the given non-zero density `nzd`. Let us see how efficient the operations can be on the matrices in sparse forms.

```
>>As = sprandn(10,10,0.2); %a sparse matrix and
>>Af = full(As); its full version
```

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

```
>>flops(0), AsA = As*As; flops %in sparse forms
   ans = 50
>>flops(0), AfA = Af*Af; flops %in full(regular) forms
   ans = 2000
>>b = ones(10,1); flops(0), x = As\b; flops
   ans = 160
>>flops(0), x = Af\b; flops
   ans = 592
>>flops(0), inv(As); flops
   ans = 207
>>flops(0), inv(Af); flops
   ans = 592
>>flops(0), [L,U,P] = lu(As); flops
   ans = 53
>>flops(0), [L,U,P] = lu(Af); flops
   ans = 92
```

Additionally, the MATLAB routine `speye(n)` is used to generate an $n \times n$ identity matrix and the MATLAB routine `spy(n)` is used to visualize the sparsity pattern. The computational efficiency of LU factorization can be upgraded if one pre-orders the sparse matrix by the symmetric minimum degree permutation, which is cast into the MATLAB routine `symmmd()`.

Interest readers are welcome to run the following program "`do_sparse`" to figure out the functions of several sparsity-related MATLAB routines.

```
%do_sparse
clear, clf
%create a sparse mxn random matrix
m = 4; n = 5; A1 = sprandn(m,n,.2)
%create a sparse symmetric nxn random matrix with non-zero density nzd
nzd = 0.2; A2 = sprandsym(n,nzd)
%create a sparse symmetric random nxn matrix with condition number r
r = 0.1; A3 = sprandsym(n,nzd,r)
%a sparse symmetric random nxn matrix with the set of eigenvalues eigs
eigs = [0.1 0.2 .3 .4 .5]; A4=sprandsym(n,nzd,eigs)
eig(A4)
tic, A1A = A1*A1', time_sparse = toc
A1f = full(A1); tic, A1Af = A1f*A1f'; time_full = toc
spy(A1A), full(A1A), A1Af
sparse(A1Af)
n = 10; A5 = sprandsym(n,nzd)
tic, [L,U,P] = lu(A5); time_lu = toc
tic, [L,U,P] = lu(full(A5)); time_full = toc
mdo = symmmd(A5); %symmetric minimum degree permutation
tic, [L,U,P] = lu(A5(mdo,mdo)); time_md=toc
```

(cf) The command 'flops' is not available in MATLAB of version 6.x and that is why we use 'tic' and 'toc' to count the process time instead of the number of floating-point operations.

APPENDIX I

MATLAB

First of all, the following should be noted:

1. The index of an array in MATLAB starts from 1, not 0.
2. A dot(.) must be put before an operator to make a termwise (element-by-element) operation.

Some of useful MATLAB commands are listed in Table I.1.

Table I.1 Commonly Used Commands and Functions in MATLAB

	General Commands
break	to exit from a for or while loop
fprintf	fprintf('\n x(%d) = %6.4f \a',ind,x(ind))
keyboard	stop execution until the user types any key
return	terminate a routine and go back to the calling routine
load *** x y	read the values of x and y from the MATLAB file ***.mat
load x.dat	read the value(s) of x from the ASCII file x.dat
save *** x y	save the values of x and y into the MATLAB file ***.mat
save x.dat x	save the value(s) of x into the ASCII file x.dat
clear	remove all or some variables/functions from memory
	Two-Dimensional Graphic Commands
bar(x,y),plot(x,y),stairs(x,y) stem(x,y),loglog(x,y) semilogx(x,y),semilogy(x,y)	plot the values of y versus x in a bar\continuous \stairs\discrete\xy-log\x-log\y-log graph

Table I.1 Commonly Used Commands and Functions in MATLAB

plot(y) (y: read-valued)	plot the values of vector\array over the index
plot(y) (y: complex-valued)	plot the imaginary part versus the real part: plot(real(y),imag(y))
bar(y, $s_1s_2s_3$) plot(y, $s_1s_2s_3$) stairs(y, $s_1s_2s_3$) stem(y, $s_1s_2s_3$) loglog(y, $s_1s_2s_3$) semilogx(y, $s_1s_2s_3$) semilogy(y, $s_1s_2s_3$) plot(y1, $s_1s_2s_3$, y2, $s_1s_2s_3$)	The string of three characters $s_1s_2s_3$, given as one of the input arguments to these graphic commands specifies the color, the symbol, and the line types: s_1(color): y(ellow), m(agenta), c(yan), r(ed), g(reen), b(lue), w(hite), (blac)k s_2(symbol):.(point), o,x,+,*, s(quare: □), d(iamond:◊), v(▽), ^(△), <(◁), >(▷), p(entagram:☆), h(exagram) s_3(line symbol): -(solid, default), :(dotted), -.(dashdot),−(dashed) (ex) `plot(x,'b+:')` plots x(n) with the + symbols on a blue dotted line
polar(theta,r)	plot the graph in polar form with the phase theta and magnitude r
Auxiliary Graphic Commands	
axis([xmin xmax ymin ymax])	specify the ranges of graph on horizontal/vertical axes
clf(clear figure)	clear the existent graph(s)
grid on/off	draw/remove the grid lines
hold on/off	keep/remove the existent graph(s)
subplot(ijk)	divide the screen into i × j sections and use the k^{th} one
text(x,y,plot(y,‘***’)	print the string ‘***’ in the position (x,y) on the graph
title(‘**’), xlabel(‘**’), ylabel(‘**’)	print the string ‘**’ into the top/low/left side of graph
Three-Dimensional Graphic Commands	
mesh(X,Y, Z)	connect the points of height Z at points (X,Y) where X,Y and Z are the matrices of the same dimension
mesh(x, y, Z)	connect the points of height Z(j, i) at points specified by the two vectors (x(i),y(j))
mesh(Z), surf(), plot3(), contour()	connect the points of height Z(j, i) at points specified by (i, j)

Once you installed MATLAB, you can click the icon like the one in the left side

to run MATLAB. Then you will see the MATLAB command window on your monitor as depicted in Fig. I.1, where a cursor appears (most likely blinking) to the right of the prompt like ‘>>’ or ‘?’ waiting for you to type in a command. If you are running MATLAB of version 6.x, the main window has not only the command window, but also the workspace box and the command history box on the left-up/down side of the command window, in which you can see the contents of MATLAB

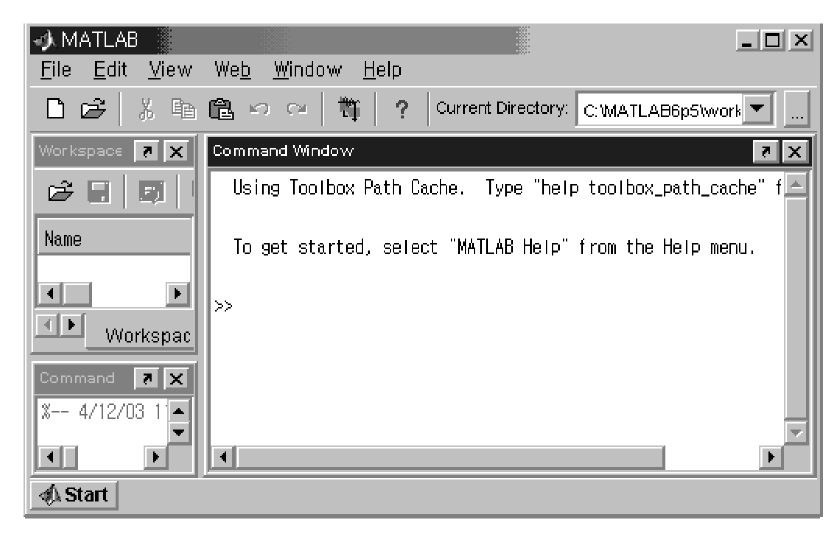


Figure I.1 The MATLAB command window with the workspace box and the command box.

memory and the commands you have typed into the Command window up to the present time, respectively. You might clear the boxes by clicking the corresponding submenu under the 'Edit' menu and even remove/restore them by un-checking/checking the corresponding submenu under the 'View' menu.

How do we work with the MATLAB command window?

- By clicking 'File' on the top menu and then 'New'/'Open' in the File pull-down menu, you can create/edit any file with the MATLAB editor.
- By clicking 'File' on the top menu and then 'Set_Path' in the File pull-down menu, you can make the MATLAB search path include/exclude the paths containing the files you want to be run.
- If you are a beginner in MATLAB, then it may be worthwhile to click 'Help' on the top menu, click 'Demos' in the Help pull-down menu, (double-)click any topic that you want to learn, and watch the visual explanation about it.
- By typing any MATLAB commands/statements in the MATLAB command window, you can use various powerful mathematic/graphic functions of MATLAB.
- If you have an m-file that contains a series of commands/statements composed for performing your job, you can type in the file name (without the extension '.m') to make it run.

It is helpful to know the procedure of debugging in MATLAB, which is summarized below.

1. With the program (you want to edit) loaded into the MATLAB Editor/ Debugger window, set breakpoint(s) at any statement(s) which you think

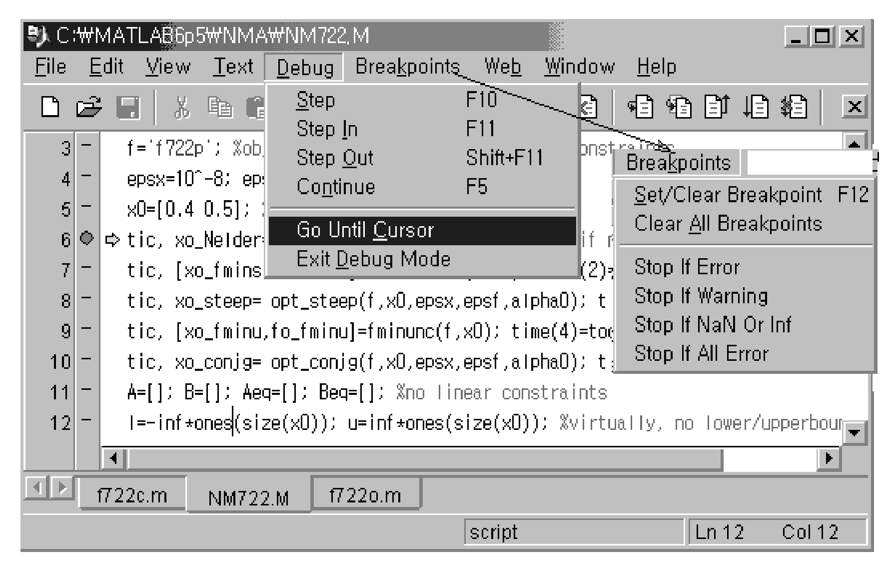


Figure I.2 The MATLAB file editor/debugger window.

is (are) suspicious to be the source(s) of error, by clicking the pertinent statement line of the program with the left mouse button and pressing the F12 key or clicking 'Set/Clear Breakpoint' in the 'Breakpoints' pull-down menu of the Editor/Debugger window. Then, you will see a small red disk in front of every statement at which you set the breakpoint.

2. Going to the MATLAB Command window, type in the name of the file containing the main program to try running the program. Then, go back to the Editor/Debugger window and you will see the cursor blinking just after a green arrow between the red disk and the first statement line at which you set the breakpoint.
3. Determining which variable to look into, go to the Command window and type in the variable name(s) (just after the prompt '`K>>`') or whatever statement you want to run for debugging.
4. If you want to proceed to the next statement line in the program, go back to the Editor/Debugger window and press the F10 (single_step) key or the F11 (step_in) key to dig into a called routine. If you want to jump to the next breakpoint, press F5 or click 'Run (Continue)' in the Debug pull-down menu of the Editor/Debugger window. If you want to run the program until just before a statement, move the cursor to the line and click 'Go Until Cursor' in the Debug pull-down menu (see Fig. I.2).
5. If you have figure out what is wrong, edit the pertinent part of the program, save the edited program in the Editor/Debugger window, and then go to the Command window, typing the name of the file containing the main program to try running the program for test. If the result seems to reflect that the program still has a bug, go back to step 1 and restart the whole procedure.

If you use the MATLAB of version 5.x, you can refer to the usage of the constrained minimization routine '`constr()`', which is summarized in the box below.

USAGE OF THE MATLAB 5.X BUILT-IN FUNCTION "CONSTR()" FOR CONSTRAINED OPTIMIZATION

```
[x,options] = constr('ftn',x0,options,l,u)
```

- Input arguments (only '`ftn`' and `x0` required, the others optional)

'`ftn`' : usually defined in an m-file and should return two output arguments, one of which is a scalar value ($f(\mathbf{x})$) of the function (`ftn`) to be minimized and the other is a vector ($\mathbf{g}(\mathbf{x})$) of constraints such that $\mathbf{g}(\mathbf{x}) \le \mathbf{0}$.

`x0` : the initial guess of solution

`options`: is used for setting the termination tolerance on $\mathbf{x}$, $f(\mathbf{x})$, and constraint violation through options(2)/(3)/(4), the number of the (leading) equality constraints among $\mathbf{g}(\mathbf{x}) \le \mathbf{0}$ through options (13), etc.
(For more details, type '`help foptions`' into the MATLAB command window)

`l,u` : lower/upper bound vectors such that $\mathbf{l} \le \mathbf{x} \le \mathbf{u}$.

- Output arguments

`x` : minimum point reached in the permissible region satisfying the constraints.

`options`: outputs some information about the search process and the result like the function value at the minimum point (`x`) reached through options (8).

REFERENCES

[B-1] Burden, Richard L., and Fairs, J. Douglas, *Numerical Analysis*, 7th ed., Brooks/Cole Thomson, Pacific Grove, CA, 2001.

[B-2] Bell, H. E., Gerschgorin's theorem and the zeros of polynomials, *Am. Math. Monthly* **72**, 292–295 (1965).

[C-1] Canale, Raymond, and Chapra, Steven, *Numerical Methods for Engineers: with Software and Programming Applications*, McGraw-Hill, New York, 2002.

[F-1] Fausett, L. V., *Applied Numerical Analysis Using MATLAB*, Prentice-Hall, Upper Saddle River, NJ, 1999.

[H-1] Hamming, R. W., *Numerical Methods for Scientists and Engineers*, 2nd ed., McGraw-Hill, New York, 1973.

[K-1] Kreyszig, Erwin, *Advanced Engineering Mathematics*, 8th ed., John Wiley & Sons, New York, 1999.

[K-2] Kreyszig, Erwin, *Introductory Functional Analysis with Applications*, John Wiley & Sons, New York, 1978.

[L-1] Lindfield, G. R., and Peny, J. E. T., *Numerical Methods Using MATLAB*, 8th ed., Prentice-Hall, Upper Saddle River, NJ, 2000.

[L-2] Luenberger, D. G., *Linear and Nonlinear Programming*, 2nd ed., Addison-Wesley Publishing Company, Reading, MA, 1984.

[M-1] Mathews, J. H., and Fink, K. D., *Numerical Methods Using MATLAB*, Prentice-Hall, Upper Saddle River, NJ, 1999.

[M-2] Maron, Melvin J., *Numerical Analysis*, Macmillan, Inc., New York, 1982.

[N-1] Nakamura, Shoichiro, *Numerical Analysis and Graphic Visualization with MATLAB*, 2nd ed., Prentice-Hall, Upper Saddle River, NJ, 2002.

[O-1] Oppenheim, Alan V., and Schafer, Ronald W., *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[P-1] Peaceman, D. W., and Rachford, H. H., The numerical solution of parabolic and elliptic differential equations, *J. Soc. Ind. Appl. Math.* **3**, 28–41 (1955).

[P-2] Pham, D. T., and Karaboga, D., *Intelligent Optimization Techniques*, Springer-Verlag, London, 1998.

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

[P-3] Phillips, C. L., and Nagle, H. T., Jr., *Digital Control System Analysis and Design*, Prentice-Hall, Upper Saddle River, NJ, 2002.

[R-1] Rao, S. S., *The Finite Element Method in Engineering*, 3rd ed., Butterworth Heinemann, Boston, 1999.

[R-2] Recktenwald, G. W., *Numerical Methods with MATLAB*, Prentice-Hall, Upper Saddle River, NJ, 2000.

[S-1] Schilling, R. J., and Harris, S. L., *Applied Numerical Methods for Engineers Using MATLAB and C*, Brooks/Cole, Pacific Grove, CA, 2000.

[S-2] Silvester, P. P., and Ferrari, R. L., *Finite Elements for Electrical Engineers*, 3rd ed., Cambridge University Press, Cambridge, U.K., 1996.

[S-3] Stoer, J., and Bulirsch, R., *Introduction to Numerical Analysis*, Springer-Verlag, New York, 1980.

[W-1] Website <http://www.maths.abdn.ac.uk/igc/testing/tch/ma2001/notes/node79 .html>

[W-2] Website <http://mathworld.wolfram.com/PoissonDistribution.html>

[W-3] Website <http://www-ccrma.stanford.edu/ jos/mdft/>

[W-4] Website <http://mathworld.wolfram.com/FourierTransform.html

[W-5] Website <http://www.psc.edu/∼burkardt/papers/linear_glossary.html>

[W-6] Website <http://www.mathworks.com/access/helpdesk/help/helpdesk.html>

[W-7] Website <http://csep1.phy.ornl.gov/CSEP/MO/NODE27.html>

[W-8] Website <http://mathews.ecs.fullerton.edu/numerical.html>

[Z-1] Zienkiewicz, O. C., and Taylor, R. L., *The Finite Element Method*, 4th ed., Vol. 1, McGraw-Hill, London, 1989.

SUBJECT INDEX

Applied Numerical Methods Using MATLAB®, by Yang, Cao, Chung, and Morris

M

N

O

P

Q

R

INDEX FOR MATLAB ROUTINES

(cf) A/C/E/P/S/T stand for Appendix/Chapter/Example/Problems/Section/Table, respectively.
(cf) The routines whose name starts with a capital letter are constructed in this book.
(cf) A program named “nmijk.m” can be found in Section i.j-k.

INDEX FOR TABLES

(cf) A: Appendix, P: Problem, S: Section, T: Table